人工智能系列

Elasticsearch大数据搜索引擎

罗　刚　编著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

Elasticsearch 搜索集群系统在生产和生活中发挥着越来越重要的作用。本书介绍了 Elasticsearch 的使用、原理、系统优化与扩展应用。本书用例子说明了 Java、Python、Scala 和 PHP 的编程 API，其中在 Java 搜索界面实现上，介绍了使用 Spring 实现微服务开发。为了扩展 Elasticsearch 的功能，本书以中文分词和英文文本分析为例介绍了插件开发方法。本书介绍了使用 Elasticsearch 作为数据管理平台的日志监控与分析方法，介绍了使用 OCR 从图像中提取文本以及问答式搜索的开发方法。

本书适用于有程序设计基础的开发人员或者对 IT 运维技术感兴趣的从业人员。

图书在版编目（CIP）数据

Elasticsearch 大数据搜索引擎 / 罗刚编著. —北京：电子工业出版社，2018.1
（人工智能系列）
ISBN 978-7-121-33233-3

Ⅰ. ①E… Ⅱ. ①罗… Ⅲ. ①搜索引擎－程序设计 Ⅳ. ①TP391.3

中国版本图书馆 CIP 数据核字（2017）第 306154 号

策划编辑：张 迪
责任编辑：底 波
印 刷：
装 订：
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 1/16 印张：14.25 字数：364.8 千字
版 次：2018 年 1 月第 1 版
印 次：2018 年 1 月第 1 次印刷
定 价：49.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）88254469，zhangdi@phei.com.cn。

前言

<<<<< PREFACE

智慧生物的大规模协作造就了惊人的进化奇迹。大规模机器集群造就机器系统进化成为强大的智能系统。Elasticsearch 作为大数据与搜索引擎技术的结合体，随着社会对大规模开源分布式搜索引擎的需求迅速成长。

由于其良好的易用性，Elasticsearch 早在 1.0 版本之前就加速了大规模搜索集群的普及。本书从基本概念开始熟悉 Elasticsearch，接下来介绍了 Elasticsearch 在 Windows 和 Linux 操作系统下的安装。除了 CURL 命令，本书还介绍了使用常用的编程工具和 Elasticsearch 搜索服务交互，包括 Java、Python、Scala 和 PHP，以及使用 SQL 语句查询 Elasticsearch 索引的方法。自然语言文本理解往往以插件形式存在于 Elasticsearch 集群中，第 2 章介绍了如何开发与测试插件。因为 Elasticsearch 经常用于实时搜索或分析，所以性能优化很重要，第 3 章介绍了如何管理 Elasticsearch 集群。为了更合理地使用和扩展 Elasticsearch，第 4 章简单分析了 github 中托管的 Elasticsearch 源代码。对于搜索引擎来说，返回结果的相关性是一个重要的话题，第 5 章讨论了这个问题。第 6 章介绍了使用 Java 开发搜索引擎 Web 用户界面的几种方法。

随着人工智能领域技术的发展，让搜索引擎智能加速变成现实。智能搜索引擎需要能够检测到并识别出图像中的文字，第 7 章介绍了结合 OpenCV 使用 Tesseract 识别文字的方法。第 8 章介绍了根据问题返回搜索结果的问答式搜索。

目前 Elasticsearch 是实时系统监控的首选，第 9 章介绍了使用 Elasticsearch 监控与分析日志，也介绍了通过物联网监控系统的方案。

本书相关的参考软件和代码在读者 QQ 群 471033528 的附件中可以找到。Elasticsearch 及其底层依赖的软件，其复杂程度已经超越了一个人所能掌握的程度。一些具体的细节也可以在读者 QQ 群中讨论。感谢早期合著者、合作伙伴、员工、学员、读者的支持，给我们提供了良好的工作基础。就像玻璃容器中的水培植物一样，这是一个持久可用的工作基础。技术的融合与创新无止境，欢迎读者一起探索。

本书适合需要具体实现搜索引擎的程序员使用，对于信息检索等相关领域的研究人员也有一定的参考价值，同时猎兔搜索技术团队已经开发出以本书为基础的专门培训课程和商业软件。

参与本书编写的还有张子宪、崔智杰、张晓斐、石天盈、张继红、张进威、刘宇、何淑琴、任通通、高丹丹、徐友峰、孙宽，在此一并表示感谢。

目录

使用 Elasticsearch

在信息时代，可供获取的数据加速涌现，我们可以通过搜索引擎来挖掘大数据的价值，百度就是一个大的数据搜索引擎。

Lucene 是一个 Java 语言开发的开源全文检索引擎工具包。Lucene 穿了一件 json 的外衣，就是 Elasticsearch。Elasticsearch 内置了对分布式集群和分布式索引的管理，所以相对 Solr 来说，更容易分布式部署。使用 Elasticsearch 的搜索系统整体架构如图 1-1 所示。

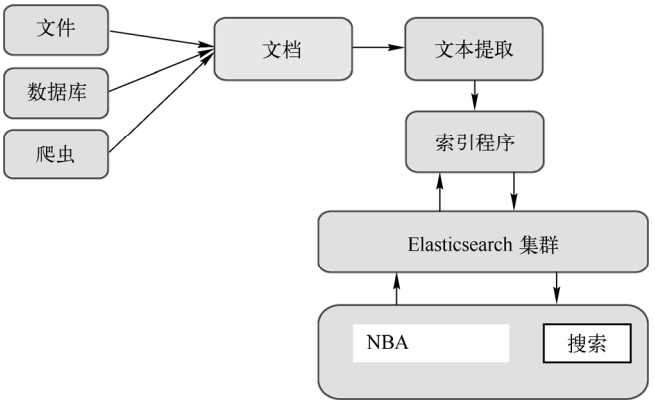

图 1-1　Elasticsearch 的外部结构

1.1 基本概念

每一个运行实例称为一个节点，每一个运行实例既可以在同一个机器上，也可以在不同的机器上。

在一个分布式系统里面，可以通过多个 Elasticsearch 运行实例组成一个集群（Cluster），这个集群里面有一个节点叫作主节点（master），Elasticsearch 是去中心化的，所以这里的主节点是动态选举出来的，不存在单点故障。

在同一个子网内，只要在每个节点上设置相同的集群名，Elasticsearch 就会自动把这些集群名相同的节点组成一个集群。节点和节点之间的通信以及节点之间的数据分配和平衡全部由 Elasticsearch 自动管理。

Elasticsearch 会把一个索引（Index）分解为多个小的索引，每一个小的索引就叫作分片（Shards）。

Elasticsearch 的每一个分片都可以有 0 到多个副本（Replicas），而每一个副本也都是分片的完整复制品，其好处是可以用它来增加速度的同时也提高了系统的容错性。

一旦 Elasticsearch 的某个节点数据损坏或服务不可用时，那么就可以用其他节点来代替坏掉的节点，以达到高可用的目的。

当有节点加入或退出时，它会根据机器的负载对索引分片进行重新分配，当“挂掉”的节点再次重新启动时也会进行数据恢复（Recovery）。

通过网关（Gateway）来管理集群恢复，可以配置集群需要加入多少个节点，才能够启动恢复。网关配置用于恢复任何失败的索引。当节点崩溃并重新启动时，Elasticsearch 将从网关读取所有索引和元数据。

Transport 代表 Elasticsearch 内部的节点或集群与客户端之间的交互方式。默认使用 TCP 协议来进行交互，同时它支持 HTTP 协议（json 格式）、Thrift、Servlet、Memcached、ZeroMQ 等多种的传输协议（通过插件方式集成）。

为了让集群能够在运行时动态附加额外的功能，使用插件机制加载实现公共接口的程序集。Elasticsearch 插件用于以各种特定的方式扩展基本的 Elasticsearch 功能。

1.2 安装

首先介绍 Elasticsearch 在 Windows 下的安装，然后介绍在 Linux 下的安装。

安装包下载网址：http://www.elasticsearch.org/download/。这里使用的版本为 5.1.2。得到文件：elasticsearch-5.1.2.zip

直接解压至某目录，例如，D:\elasticsearch-5.1.2。下载完成解压后有以下几个路径：bin 是运行的脚本，config 是设置文件，lib 中放依赖的包。

到目录 D:\elasticsearch-5.1.2\bin 下，运行 elasticsearch.bat。

如果显示 Java 虚拟机内存不够，则可以在 D:\elasticsearch-5.1.2\config\jvm.options 配置文件中调整内存大小。

成功启动 Elasticsearch 后，在浏览器中打开网址：http://localhost:9200/。

启动成功后，会在解压目录下增加两个文件夹：data 用于存储索引数据，logs 用于日志记录。因为创建索引耗时，所以将文档预先写入一个日志目录中。

通过 HTTP 协议发送指令来和 Elasticsearch 交互。curl 是一个知名的网络命令行工具，可以用来发送 GET 或 POST 命令。在 Linux 下默认已经安装了这个命令行工具，但也有 Windows 版本的 curl，下载地址为 http://www.paehl.com/open_source/，这是一个编译好的 curl.exe 文件。需要在 Windows 命令行下运行这个工具。

默认情况下，Elasticsearch 的 RESTful 服务只有本机才能访问，也就是说无法从主机访问虚拟机中的服务。为了方便调试，可以修改 config/elasticsearch.yml 文件，加入以下两行：

```
http.host: 0.0.0.0
transport.host: 127.0.0.1
```

但线上环境切忌不要这样配置，否则任何人都可以通过这个接口修改 ES 的数据。

为了看到索引内容，需要安装 head 插件。Elasticsearch 5.x 安装 head 插件需要随同 NodeJS 一起安装的包管理工具 npm。

在 Linux 下首先安装 JDK。

```
#wget -c --header "Cookie: oraclelicense=accept-securebackup-cookie" http://download.oracle.com/otn-pub/java/jdk/8u131-b11/d54c1d3a095b4ff2b6607d096fa80163/jdk-8u131-linux-x64.rpm
# rpm -i ./jdk-8u131-linux-x64.rpm
```

验证 Java 安装：

```
#java –version
```

输出如下：

```
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

启动时可能会出现一些错误，根据需要增加打开文件和进程的数量以及虚拟内存数量。

编辑 limits.conf：

```
vi /etc/security/limits.conf
```

添加如下内容：

```
* soft nofile 65536
* hard nofile 131072
* soft nproc 2048
* hard nproc 4096
```

编辑 90-nproc.conf。

```
vi /etc/security/limits.d/90-nproc.conf
```

修改如下内容：

```
* soft nproc 1024
```

修改为：

```
* soft nproc 2048
```

修改配置文件 sysctl.conf：

```
vi /etc/sysctl.conf
```

添加下面配置：

```
vm.max_map_count=655360
```

执行命令：

```
sysctl -p
```

在 Linux 下不能以 root 用户启动 Elasticsearch，所以需要先创建用户。

```
# adduser ops
```

设置密码：

```
# passwd ops
```

下载并解压缩 elasticsearch-5.1.2.tar.gz。启动：

```
# sh elasticsearch
```

执行 elasticsearch 默认会以控制台的方式执行，如果想脱离控制台运行，加上参数 -d 即可（./elasticsearch -d）。

对于旧版本的 Linux，启动时会提示警告：unable to install syscall filter，可以忽略这个警告。这个过滤器是 Elasticsearch 安全模块做出了额外的努力，撤销 Linux 进程权限，减少“攻击载体”恶意活动。

使用 HTTP 请求与 Elasticsearch 打交道。HTTP 请求包括请求的 URL 地址、HTTP 命令（GET、POST 等）等。为了简洁而一致地描述 HTTP 请求，Elasticsearch 文档使用 cURL 命令行语法。这也是在用户社区中描述对 Elasticsearch 的请求的标准做法。通过 cURL 命令发送 HTTP 请求给本地节点的例子如下。


```
# curl -XGET 'http://localhost:9200/'
```

使用 cURL 的简单搜索请求：

```
# curl -XPOST "http://localhost:9200/_search" -d'
    {
        "query": {
          "match_all": {}
        }
    }'
```

上述代码片段在控制台中执行时，使用 3 个参数运行 cURL 程序。第 1 个参数-XPOST 意味着 cURL 所做的请求应使用 HTTP 的 POST 请求；第 2 个参数 http://localhost:9200/_search 是请求的 URL；第 3 个参数-d'{...}'使用-d 来指示 cURL 发送跟随这个标记的 HTTP POST 数据。

可以用字符浏览器 links 访问：

```
# links http://localhost:9200/
```

head 插件是 Elasticsearch 集群的 Web 前端，作为一个单独的 Web 应用运行。在 Linux 下安装 head 插件的过程如下。

首先安装 npm。npm 是一个 node 包管理和分发工具，安装命令如下。

```
# yum install npm
```

然后下载 elasticsearch-head。用 git 命令复制一个小的本地仓库。

```
# git clone git://github.com/mobz/elasticsearch-head.git
```

```
# cd elasticsearch-head
```

安装包：

```
# npm install
```

启动：

```
# npm run start
```

为了避免出现跨域问题，在文件 elasticsearch.yml 中添加如下配置。

```
http.cors.enabled: true
http.cors.allow-origin: "*"
```

通过 CURL 发送 POST 请求停止节点。停止本地节点：

```
# curl -XPOST 'http://localhost:9200/_cluster/nodes/_local/_shutdown'
```

停止集群中所有的节点：

```
# curl -XPOST 'http://localhost:9200/_shutdown'
```


1.3 搜索集群

分布式索引需要考虑的问题如下。

- 确定一个集群中应包括哪些机器。
- 数据如何分布。
- 快速的分布式打分。

数据分布有两种方法：按文档分片或按词分片。不同的文档集合放在不同的机器上，不同的词列表集合放在不同的机器上。Elasticsearch 仍然是把不同的文档集合放在不同的机器上。

最简单的方法是手工管理集群。假设要手工建立两台机器组成的集群。修改 elasticsearch.yml 文件如下。

对于主机 esa.lietu.com：

```
cluster.name: OurCluster
node.name: "esa"
discovery.zen.ping.multicast.enabled: false
discovery.zen.ping.unicast.hosts: ["esb.lietu.com"]
```

对于主机 esb.lietu.com：

```
cluster.name: OurCluster
node.name: "esb"
discovery.zen.ping.multicast.enabled: false
discovery.zen.ping.unicast.hosts: ["esa.lietu.com"]
```

一般使用多播自动建立集群。只要其他节点与第一个节点有相同的 cluster name，它

就自动发现并加入第一个节点的集群。

1.4 创建索引

表的结构和相关设置的信息在 mapping 中设置。

索引模板也可以放在模板目录下的配置位置（path.conf）。注意，所有有主节点资格的节点都要放。例如，一个叫作 template_1.json 的文件可以放在 config/templates 目录下，如果它能够匹配一个索引，就会把它添加进去。放在 config/[index_name]/[some_name].json 的例子如下。

```
{
    "[type]" : {
        "properties" : {
            "title" : {
                "type" : "string",
                "boost" : 2.0
            }
            "tags": {
                "type" : "string"
            }
        }
    }
}
```

在命令行设置 Mapping 的例子如下。

```
curl -XPOST localhost:9200/wf_mds_org（索引名称） -d'{
    "settings": {
        "number_of_shards": 1,
        "number_of_replicas": 0,
        "index.refresh_interval": "-1",
        "index.translog.flush_threshold_ops": "100000"
    },
    "mappings": {
        "org": {    //(类型)
            "_all": {
                "analyzer": "ike"
            },
            "_source": {
                "compress": true
            },
            "properties": {
                "_ID": {
                    "type": "string",
                    "include_in_all": true,
                    "analyzer": "keyword"
```

```
},
                    "NAME": {
    "type": "multi_field",
    "fields": {
        "NAME": {
            "type": "string",
            "analyzer": "keyword"
        },
        "IKO": {
            "type": "string",
            "analyzer": "ike"
        }
    }
},
                    "SHORTNAME": {
    "type": "string",
    "index_analyzer": "pct_spliter",
    "search_analyzer": "keyword",
    "store": "no"
},
                    "OLDNAME": {
    "type": "multi_field",
    "fields": {
        "OLDNAME": {
            "type": "string",
            "analyzer": "keyword"
        },
        "IKO": {
            "type": "string",
            "analyzer": "ike"
        }
    }
},
                    "TNAME": {
    "type": "string",
    "analyzer":"custom_snowball_analyzer",
    "store": "no"
},
                    "TSNAME": {
    "type": "string",
    "index": "no",
    "store": "no"
},
                    "TONAME": {
    "type": "string",
    "index": "no",
```

```
                    "store": "no"
                }
            }
        }
    }
}'
```

上面给出了一个完整的 Mapping，可将 Mapping 信息大致分成 settings 和 mappings 两部分，settings 主要作用于 index 的一些相关配置信息，如分片数、副本数等，以及 tranlog 同步条件、refresh 条等。Mappings 部分主要是索引结构的一些说明。mappings 主体上大致又分成_all、_source、properites 这 3 部分。

（1）_all：主要指的是 All Field 字段，我们可以将一个或多个包含进去，在进行检索时无须指定字段的情况下检索多个字段。前提是要开启 All Field 字段：

```
"_all" : {"enabled" : true}
```

（2）_source：主要指的是 Source Field 字段 Source 可以理解为 Es 除了将数据保存在索引文件中，另外还有一分源数据。_source 字段我们在进行检索时相当重要，如果在{"enabled" : false}情况下默认检索只会返回 ID，需要通过 Fields 字段到倒排索引中去取数据，当然效率不是很高。如果觉得 enabale:true 时，索引的膨胀率比较大的情况下可以通过下面一些辅助设置进行优化。

Compress:是否进行压缩，建议一般情况下将其设为 true。

```
"includes" : ["author", "name"],
"excludes" : ["sex"]
```

上面的 includes 和 excludes 主要是针对在默认情况下面_source 一般是保存全部批量导入的数据，我们可以通过 include、excludes 在字段级别上做出一些限索。

（3）properites 部分是最重要的，主要是针对索引结构和字段级别上面的一些设置。

```
"NAME": { //字段项名称对应 lucene 里面的 FiledName
        "type": "string",    //type 为字段项类型
        "analyzer": "keyword"//字段项分词的设置对应 Lucene 里面的 Analyzer
        },
```

在 Es 中字段项的 type 是一个很重要的概念，Es 中在 Lucene 的基础上提供了比较多的类型，而这些类型对应一些相关的检索特性，如 Date 型，可以使用 [2001 TO 2012]的方式进行范围检索等，Elasticsearch 的类型如下。

简单类型：

string：字符型最常用的。

integer：整型。

long：长整型。

float：浮点型。

double：双字节型。

boolean：布尔型。

复杂类型：

Array：数组型。

```
"lists":{{"name":"…"},{"name":"…"}}
```

Object：对象类型。

```
"author":{"type":"object","perperites":{"name":{"type":"string"}}}
```

Multi_field：多分词字段，针对一个字段提供多种分词方式。

Nested：嵌入类型使用的较多。

Analyzer 在 Lucene 中是一个分词器的概念，我们知道 Es 是建立在 Lucene 之上的，所以这里的 Analzyer 同样也适用，Mapping 中的 Analyzer 主要是指定字段采用什么分词器，具体的程序和配置分词在插件和配置都有过一些说明。

Analyzer 在 Es 中分为 index_analyzer 和 search_analyzer。

Index_analzyer 指的是索引过程中采用的分词器。

Search_analyzer 指的是检索过程中采用的分词器。

我们知道 index 和 search 是两个过程，但要尽量保证这两个过程和分词方式一致，这样可以保证查全和查准，否则再好的分词，index 和 search 采用的不相同也是无用的。

与 analyzer 相关的就是另外一个 index 项。例如：

```
"HC":{ "type":"string", "index":"no", "store":"no"}
```

index 表示该字段是否索引，如果 index 为 no，那么 analyzer 设置为什么都没用。

最后是 store 项，store 项表示该项是否存储到索引中去，并不是_source。

可以查询所有的文档类型或只选择单一的文档类型来查询。

自定义的分词器。修改配置文件，位于 config 目录下的 elasticsearch.yml。

如何指定不同的列用不同的分词器？调用 http://localhost:9200/index_name/_mapping 这样的接口提供了列所使用的分析器。例如：

```
http://localhost:9200/news/_mapping
```

返回结果：

```
{"news":{"mappings":{"type1":{"properties":{"body":{"type":"string"},"title":{"type":"string"}}}}}}
```

1.5　使用 Java 客户端接口

虽然可以用 curl 这样的命令行工具和 Elasticsearch 搜索服务器打交道，但实际开发中一般使用 Elasticsearch 的 Java API。

TransportClient 使用 transport 模块远程连接到 Elasticsearch 集群。首先保证客户端的 JDK 和服务器的版本一致。然后在 Eclipse 中创建一个 Maven 项目，或者把 D:\elasticsearch-5.1.2\lib 目录下的 jar 包复制到 lib 目录。Elasticsearch 的 Java 客户端 API 最少依赖四个包：transport-5.3.0.jar、elasticsearch-5.3.0.jar、joda-time-2.9.5.jar 和 lucene-core-6.4.1.jar。

如果使用 maven，则需要添加依赖关系：

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>transport</artifactId>
    <version>5.3.0</version>
</dependency>
<!-- Log Dependencies required by elasticsearch 5 -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.7</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.7</version>
</dependency>
```


使用这个 maven 配置会得到 52 个 jar 文件。可以在安装 Maven 后，通过命令得到这些 jar 包。

```
>mvn -f pom.xml dependency:copy-dependencies
```

通过 TransportClient 这个接口和 Es 集群进行通信时，需要指定 Es 集群中其中一台或多台机器的 IP 地址和端口号。连接本地集群的例子代码如下。

```
TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
        .addTransportAddress(new InetSocketTransportAddress(InetAddress
                .getByName("192.168.10.150"), 9200));
```

指定多个机器连接的例子如下。

```
Client client = new PreBuiltTransportClient(Settings.EMPTY)
        .addTransportAddress(
                new InetSocketTransportAddress(InetAddress
                        .getByName("host1"), 9300))
        .addTransportAddress(
                new InetSocketTransportAddress(InetAddress
                        .getByName("host2"), 9300));
//使用连接…
client.close();
```

集群名默认是 Elasticsearch。如果需要更改集群名，需要如下设置：

```
Settings settings = Settings.builder()
        .put("cluster.name", "ElasticSearchClusterName").build();
Client client = new PreBuiltTransportClient(settings);
```

可以设置 client.transport.sniff 为 true 来使客户端去窥探整个集群的状态，把集群中其他机器的 IP 地址加到客户端中，这样做的好处是不用手动设置集群里所有集群的 IP 连接到客

户端，它会自动帮你添加，并且自动发现新加入集群的机器，代码实例如下。

```
Settings settings = Settings.builder()
            .put("client.transport.sniff", true).build();
TransportClient client = new PreBuiltTransportClient(settings);
```

客户端连接 Elasticsearch 耗时很长，优化的办法是用单例，不用释放连接。使用静态内部类实现单例模式：

```
public class ClientFactory {

    public static Client create(){
        Settings settings = Settings.builder()
                .put("client.transport.sniff", true).build();
        TransportClient client = new PreBuiltTransportClient(settings);

        return client;
    }
}

public class ClientBuilder {

    private static class StaticHolder {
        static final Client INSTANCE = ClientFactory.create();
    }

    public static Client getSingleton() {
        return StaticHolder.INSTANCE;
    }
}
```

当首次调用 getSingleton()方法时，访问静态内部类中的静态成员变量，此时该内部类需要调用工厂类创建一个唯一的实例。以后再次访问 getSingleton()方法时，会直接返回 INSTANCE。

1.5.1　创建索引

创建索引：

```
CreateIndexRequestBuilder createIndexRequestBuilder =
    client.admin().indices().prepareCreate("test1");   //创建一个称为 test1 的索引
CreateIndexResponse response = createIndexRequestBuilder.execute().actionGet();
System.out.println(response.isAcknowledged());
```

使用 json 格式定义索引库的结构。为了方便生成 json 串，需要导入静态方法 XContentFactory.jsonBuilder()。

```
import static org.elasticsearch.common.xcontent.XContentFactory.jsonBuilder;
```

可以在 CreateIndexRequestBuilder.addMapping()方法中定义字段的类型：

```
Settings settings = Settings.builder().loadFromSource(jsonBuilder()
        .startObject()
          .startObject("analysis")
            .startObject("analyzer")
              .startObject("synonym")
                .field("tokenizer", "whitespace")
                .field("filter", new String[] {"synonym"})
              .endObject()
            .endObject()
            .startObject("filter")
              .startObject("synonym")
                .field("type", "synonym")
                .field("synonyms", new String[] {"Highlandstreet, ravioli"})
                .field("ignore_case", true)
                .field("expand", true)
              .endObject()
            .endObject()
          .endObject()
        .endObject().string(), XContentType.JSON).build();

XContentBuilder content = jsonBuilder()
    .startObject()
      .startObject("properties")
        .startObject("title")
          .field("type", "string")
          .field("analyzer", "synonym")
        .endObject()
        .startObject("location")
          .field("type", "geo_point")
        .endObject()
      .endObject()
    .endObject();

CreateIndexResponse response =
    node.client().admin().indices().prepareCreate("index_name").setSettings(settings)
      .addMapping("mapping_name1", content).get();
```

假设 json 文件中存储了索引的映射/设置，那么可以用如下代码设定索引库结构。

```
CreateIndexRequestBuilder createIndexRequestBuilder =
          client.admin().indices().prepareCreate(index);
String mapping_json = new String(Files.readAllBytes(json_mapping_path));
createIndexRequestBuilder.addMapping("my_mapping", mapping_json);
CreateIndexResponse indexResponse =
```

```
        createIndexRequestBuilder.execute().actionGet();
```

也可以使用 PutMappingRequest 设置索引库结构。首先定义：

```
private static XContentBuilder getCnMapping(String indexType)
            throws Exception {
    XContentBuilder mapping = XContentFactory.jsonBuilder().startObject()
                .startObject(indexType)
                .startObject("properties")

                .startObject("repname").field("type", "string")
                .field("store", "yes").field("analyzer", "cn") //词
                .endObject()

                .startObject("repnameS").field("type", "string")
                .field("store", "yes").field("analyzer", "standard") //字
                .endObject()

                .startObject("path").field("type", "string")
                .field("store", "yes").field("analyzer", "standard")
                .endObject()

                .startObject("contents").field("type", "string")
                .field("store", "yes").field("analyzer", "cn") //词
                .endObject()

                .startObject("contentsS").field("type", "string")
                .field("store", "yes").field("analyzer", "standard") //字
                .endObject()

                .endObject() //end properties
                .endObject() //end index type
                .endObject();
    return mapping;
}
```

然后调用 IndicesAdminClient.putMapping()方法设定索引库结构：

```
PutMappingRequest mappingRequest = Requests
            .putMappingRequest(indexName).type(type).source(mapping);
PutMappingResponse putMappingResponse = client.admin().indices()
            .putMapping(mappingRequest).actionGet();
```

使用 IndicesAdminClient 删除索引。

```
IndicesAdminClient admin = client.admin().indices();
admin.prepareDelete(indexName).execute().actionGet().isAcknowledged();
```

使用 IndicesAdminClient.prepareCreate()方法设置好创建索引需要的参数。

```
//首先创建索引库
CreateIndexResponse indexresponse = client.admin().indices()
            //这个索引库的名称还必须不包含大写字母
            .prepareCreate("testindex").setSettings(settings).execute()
            .actionGet();
System.out.println(indexresponse.isAcknowledged()); //看是否成功创建索引
```

CreateIndexRequestBuilder.setSettings()方法设置更多的参数：

```
IndicesAdminClient ac = client.admin().indices();
String index="test1";
CreateIndexRequestBuilder builder = ac.prepareCreate(index);

Builder setting = Settings.builder().put("number_of_shards", 1);
builder.setSettings(setting);
```

1.5.2 增加、删除与修改数据

可以单条或批量插入数据。插入单条记录的方法：

```
public void insert(String index, String type, Map<String, String> data) {
    Gson gson = new Gson();
    String json = gson.toJson(data);   //使用 Gson 把 Map 转换成 json 字符串
    Client client = getClient();
    IndexRequestBuilder indexBuilder = client.prepareIndex(index, type)
            .setSource(json);
    IndexResponse response = indexBuilder.execute().actionGet();
}
```

索引一个 json 字符串表示的文档。

```
String json = "{" +
        "\"user\":\"kimchy\"," +
        "\"postDate\":\"2013-01-30\"," +
        "\"message\":\"trying out Elasticsearch\"," +
    "}";

IndexResponse response = client.prepareIndex("twitter", "tweet")
        .setSource(json)
        .execute()
        .actionGet();
```

XContentFactory 是 Elasticsearch 内部的一个工具类。使用这个类得到 json 格式的字符串。

```
XContentBuilder b = jsonBuilder().startObject();
b.field("title", "标题");
b.field("body", "内容");
```

```
b.endObject();

String id="http://xxx";   //唯一列的值

IndexRequestBuilder irb = client.prepareIndex(getIndexName(), getIndexType(), id).
            setSource(b);
irb.execute().actionGet();
```

批量插入数据：

```
public void bulkInsert(String index, String type, Map sourceMap) {
    Client client = getClient();
    BulkRequestBuilder bulkRequestBuilder = client.prepareBulk();
    bulkRequestBuilder.add(client.prepareIndex(index, type).setSource(
            sourceMap));
    BulkResponse bulkResponse = bulkRequestBuilder.execute().actionGet();
}
```

批量插入数据可能出错：

```
if (bulkResponse.hasFailures()) {
  this.logger.error(bulkResponse.buildFailureMessage());
}
```


ElasticSearchClient 类简单封装了增加删除文档的方法。

```
ElasticSearchClient esc = new ElasticSearchClient();
String index="test1";   //索引名
String type="type1";   //类型，类似表名

Map<String , String> sourceMap = new HashMap<String , String>();
sourceMap.put("title", "value1");   //列名是 title，值是 value1
sourceMap.put("body", "value2");     //列名是 body，值是 value2
esc.insertRecord(index, type, sourceMap);
```

插入多值列：

```
String[] cn = { "意义 1", "意义 2"};
XContentBuilder qBuilder = jsonBuilder().startObject();
qBuilder.field("word", "hidden");
qBuilder.field("pos", "adj");
qBuilder.startArray("cn");
for (String field : cn) {
    qBuilder.value(field);
}
qBuilder.endArray();
qBuilder.endObject();

IndexRequestBuilder irb = client.prepareIndex(getIndexName(),
            getIndexType()).setSource(qBuilder);
```

通过重试解决 no node available 错误：

```
while (true) {
        try {
                bulk.execute().actionGet(getRetryTimeout());
                break;
        }
        catch (NoNodeAvailableException cont) {
                Thread.sleep(5000);
                continue;
        }
}
```

如果要存储大的二进制文件，可以使用 attachments 插件。

删除单条数据：

```
public static void DeleteResponse(Client client){
        DeleteResponse reponse = client.prepareDelete("twitter","tweet","AVbFE").get();
        System.out.println(reponse.isFound());
}
```


修改数据：

```
UpdateRequest updateRequest = new UpdateRequest();
updateRequest.index("twitter1");//索引名
updateRequest.type("tweet");//类型
updateRequest.id("2");//id
updateRequest.doc(jsonBuilder()
                .startObject()
                        .field("user", "john")
                .endObject());
client.update(updateRequest).get();
```

1.5.3 分析器

一段有意义的文字需要通过分析器分割成一个个词语后才能按关键词搜索。使用客户端接口测试分析器：

```
AnalyzeRequest analyzeRequest = new AnalyzeRequest();
analyzeRequest.text("中华人民共和国");
ActionFuture<AnalyzeResponse> analyzeResponseActionFuture =
          client.admin().indices().analyze(analyzeRequest);
List<AnalyzeResponse.AnalyzeToken> analyzeTokens =
         analyzeResponseActionFuture.actionGet().getTokens();
for (AnalyzeResponse.AnalyzeToken analyzeToken    : analyzeTokens){
        System.out.println(analyzeToken.getTerm());
}
```

1.5.4　数据导入

对于不同的数据源，使用不同的工具将数据导入到 Elasticsearch。

- go-mysql-elasticsearch（https://github.com/siddontang/go-mysql-elasticsearch）是一个将 MySQL 数据同步到 Elasticsearch 的服务。它首先使用 mysqldump 来获取原始数据，然后用 binlog 增量同步数据。
- mongo-connector（https://github.com/mongodb-labs/mongo-connector）从 MongoDB 集群创建一个管道，写入到一个或多个目标系统，如 Solr，Elasticsearch 或另一个 MongoDB 集群。它将 MongoDB 中的数据同步到目标，然后追加 MongoDB 的日志 oplog，与 MongoDB 的实时操作保持一致。

1.5.5　通过摄取快速导入数据

摄取（Ingest）是 Elasticsearch 5.0 中的一个新功能，它可以在将数据注入 Elasticsearch 时即时转换数据。

使用摄取，可以：

- 定义管道；
- 模拟一条管道；
- 在索引时使用管道即时转换数据。

管道用于定义可以在源文档上执行的所有操作。每个操作由处理器执行。处理器基本上将文档作为输入，然后转换为文档的新版本。

已经有很多处理器，如下。

- Date 处理器：从字段中解析出日期，然后使用这个日期或时间戳作为文档的时间戳。
- KV 处理器：把消息转换成键值对。
- Lowercase 处理器：把字符串小写化。
- Uppercase 处理器：把字符串大写化。

定义一个摄取管道小写化的例子如下。

```
BytesReference source = jsonBuilder().startObject()
            .field("description", "my_pipeline").startArray("processors")
            .startObject().startObject("lowercase").endObject().endObject()
            .endArray().endObject().bytes();
PutPipelineRequest request = new PutPipelineRequest("1",
            source,
            XContentType.JSON);
client.admin().cluster().putPipeline(request).get();
```

1.5.6　索引库结构

在 Mapping 中声明索引库的结构。设置索引库的结构的代码如下。

```
IndicesAdminClient ac = client.admin().indices();
CreateIndexRequestBuilder builder = ac.prepareCreate(index).setSettings(setting);

String type = "type1";
builder = builder.addMapping(type, getMapping());
CreateIndexResponse indexresponse = builder.execute().actionGet();
```

1.5.7 查询

Elasticsearch 提供基于 json 的 Query DSL 查询表达式，DSL 即领域专用语言。可以把 Query DSL 当作一系列的抽象的查询表达式树。某个查询能够包含其他查询（如布尔查询），有些查询能够包含过滤器（如 constant_score），还有的可以同时包含查询和过滤器（如 filtered）。

Query DSL 是一个通用的查询框架。可以通过 Java API 向搜索服务器发送 json 格式的 Query DSL。

基本词查询：


```
String keyWords = "value1";   //查询词
TermQueryBuilder qb=new TermQueryBuilder("title", keyWords);
//或者用 QueryBuilder qb = QueryBuilders.termQuery("title", keyWords);

String index = "test1"; //索引名

Client client = getClient();
SearchResponse searchResponse =
                    client.prepareSearch(index).setQuery(qb).execute().actionGet();
```

返回结果：

```
SearchHits hits = response.getHits();
for (SearchHit hit : hits) {
      System.out.println("id "+hit.getId()); //文档 ID
      Map<String, Object> result = hit.getSource(); //键是列名，值是文档中该列的值
      for (final Entry<String, Object> entry : result.entrySet()) {
            System.out.println(entry.getKey() + " : " + entry.getValue());
      }
}
```

如果不指定列，则可以通过 SearchHit.getSource()方法返回所有列的值，也可以调用 setFetchSource()方法只返回指定查询的列，如只返回 word 列的内容：

```
SearchResponse searchResponse = client.prepareSearch(DictCrawler.getIndexName())
            .setQuery(query)
          .setFetchSource(new String[]{"word"},null).execute().actionGet();
SearchHits hits = searchResponse.getHits();
for (SearchHit hit : hits) {
```

```
        Map<String, SearchHitField> map =    hit.getFields();

        SearchHitField field = map.get("word");
        System.out.println(field.value());
    }
```

可以指定索引库中的类型：

```
String index = "test1"; //索引名
SearchRequestBuilder builder = client.prepareSearch(index);
String type = "type1"; //类型名
builder.setTypes(type);
```

短语查询：

```
String keyWords = "自 2006 年 8 月 2 日北京市";
String field = "body";
MatchQueryBuilder qb = QueryBuilders.matchPhraseQuery(field, keyWords);
```

使用 BoolQueryBuilder 把多个查询条件串联起来，如同时查询标题和内容列：

```
String keyWords = "工作顺利完成";
String field = "body";
MatchQueryBuilder pqBody = QueryBuilders.matchPhraseQuery(field, keyWords);

field = "title";
MatchQueryBuilder pqTitle = QueryBuilders.matchPhraseQuery(field, keyWords);

QueryBuilder qb = QueryBuilders.boolQuery().should(pqBody).should(pqTitle);
```

同时查询多列：

```
MultiMatchQueryBuilder qb = QueryBuilders.multiMatchQuery(keyWords, "title","body");
```

为了把连续匹配的文档排在前面，也可以返回不连续匹配的文档，组合模糊匹配和短语查询。

```
//让连续出现查询串的文档相关度高
String field = "body";
MatchQueryBuilder pqBody = QueryBuilders.matchPhraseQuery(field, keyWords);

field = "title";
MatchQueryBuilder pqTitle = QueryBuilders.matchPhraseQuery(field, keyWords);
//可以匹配上不连续出现查询串的文档
QueryStringQueryBuilder fuzzyQb = new QueryStringQueryBuilder(keyWords);

//用 OR 关系连接多个查询条件
QueryBuilder qb =
        QueryBuilders.boolQuery().should(pqBody).should(pqTitle).should(fuzzyQb);
```

测试得到的 Query DSL：

```
String keyWords = "自 2006 年 8 月 2 日北京市";
String field = "body";
MatchQueryBuilder pqBody = QueryBuilders.matchPhraseQuery(field, keyWords);

field = "title";
MatchQueryBuilder pqTitle = QueryBuilders.matchPhraseQuery(field, keyWords);

QueryStringQueryBuilder fuzzyQb = new QueryStringQueryBuilder(keyWords);

QueryBuilder qb =
  QueryBuilders.boolQuery().should(pqBody).should(pqTitle).should(fuzzyQb);
System.out.println(qb);
```

输出结果：

```
{
  "bool" : {
    "should" : [ {
      "match" : {
        "body" : {
          "query" : "自 2006 年 8 月 2 日北京市",
          "type" : "phrase"
        }
      }
    }, {
      "match" : {
        "title" : {
          "query" : "自 2006 年 8 月 2 日北京市",
          "type" : "phrase"
        }
      }
    }, {
      "query_string" : {
        "query" : "自 2006 年 8 月 2 日北京市"
      }
    } ]
  }
}
```

结果中只返回指定的几列。

在 http://localhost:9200/_plugin/head/运行：

```
{
  "query": {
    "match_all": {}
  },
```

```
    "fields": ["_id"]
}
```

通过 Java 客户端执行：

```
            String json = "{\n" +
                    "    \"query\" : {\n" +
                    "        \"match_all\" : {}\n" +
                    "    },\n"+
                    "    \"fields\" : [\"_id\"]\n" +
                "}";
            System.out.println(json);
            Client client = getClient();
            SearchResponse searchResponse =
            client.prepareSearch(ESConfig.indexName)
                        .setSource(json).execute().get();
            SearchHits hits = searchResponse.getHits();
```



```
String text = "地质"; //查询词
String[] fields = { "repname", "repsubdate" };

//生成 json 内容
XContentBuilder qBuilder = XContentFactory.jsonBuilder().startObject();

qBuilder.startObject("query");
qBuilder.startObject("term");
qBuilder.field("repname", text);
qBuilder.endObject();
qBuilder.endObject();
//指定返回列
qBuilder.startArray("fields");
for (String field : fields) {
    qBuilder.value(field);
}
qBuilder.endArray();
qBuilder.endObject();

//生成的查询 DSL {"query":{"term":{"repname":"地质"}},"fields":["repname","repsubdate"]}
System.out.println(qBuilder.string());

Client client = getClient();
SearchResponse searchResponse = client
            .prepareSearch(ESConfig.indexName).setSource(qBuilder)
            .execute().actionGet();
SearchHits hits = searchResponse.getHits();
```

```
long totalHits = hits.getTotalHits(); //得到结果总数
System.out.println("totalHits:" + totalHits);

for (SearchHit hit : hits) {
        SearchHitField j = hit.field("repname");
        System.out.print(j.value());
        j = hit.field("repsubdate");
        System.out.println("    " + j.value());
        System.out.println("score: " + hit.getScore());
}
```

在客户端输入 json 查询串：

```
String queryString="我爱北京天安门";
System.out.println(QueryBuilders.queryString(queryString).analyzer("cn"));
```

输出：

```
{
  "query_string" : {
    "query" : "我爱北京天安门",
    "analyzer" : "cn"
  }
}
```


1.5.8 区间查询

在商品搜索中，经常需要指定按时间条件或价格等数值条件查找。
假设有一个索引：

```
curl -XPUT localhost:9200/test
```

还有一些文档：

```
curl -XPUT localhost:9200/test/range/1 -d '{"age": 9}'
curl -XPUT localhost:9200/test/range/2 -d '{"age": 12}'
curl -XPUT localhost:9200/test/range/3 -d '{"age": 16}'
```

在一定范围内查询这些文档：

```
curl -XGET 'http://localhost:9200/test/range/_search?pretty=true' -d '
{
    "query" : {
        "range" : {
            "age" : {
                "from" : "10",
                "to" : "20",
```

```
                "include_lower" : true,
                "include_upper": true
            }
        }
    }
}
'
```

这将会返回文档 2 和文档 3。
没有 json 请求的方法：

```
curl -XGET --globoff 'localhost:9200/test/range/_search?q=age:["10"+TO+"20"]&pretty=true'
```

可以通过 RangeQueryBuilder 来实现这样的时间条件区间条件查找：

```
RangeQueryBuilder queryDate = QueryBuilders.rangeQuery("age").to(to).from(from);
```

1.5.9　排序

可以在指定字段上排序：

```
SearchRequestBuilder searchRequestBuilder = client.prepareSearch(index);

FieldSortBuilder sorter = new FieldSortBuilder("price");
searchRequestBuilder.addSort(sorter);
```

按多列排序的例子：

```
GET /my_index/my_type/_search
{
    "sort" : [
        { "post_date" : {"order" : "asc"}},
        "user",
        { "name" : "desc" },
        { "age" : "desc" },
        "_score"
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

1.5.10　分布式搜索

共有两种查询类型。

- QUERY_THEN_FETCH:

主节点将请求分发给所有分片，各个分片打分排序后将数据的 ID 和分值返回给主节点，主节点收到后进行汇总排序再根据排序后的 ID 到对应的节点读取对应的数据后返回给客户端，此种方式需要和 Es 交互两次。这种方式是 Es 的默认查询方式。

- DFS_QUERY_THEN_FETCH

将各个分片的规则统一起来进行打分，会多一次初始扫描过程。使用这种查询的代码如下。

```
esClient.prepareSearch(ElasticSearchClient.getIndexName())
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH);
```

1.5.11 过滤器

如果只有一个过滤器，可以只使用 match_all 查询搭配这个过滤器。

```
MatchAllQueryBuilder matchQqueryBuilder = QueryBuilders.matchAllQuery();

//Add filter on the query based on filtered query
AndFilterBuilder andFilterBuilder = FilterBuilders.andFilter();
andFilterBuilder.add(FilterBuilders.termFilter("version", "3"));
requestBuilder.setQuery(QueryBuilders.filteredQuery(matchQqueryBuilder,
                        andFilterBuilder));
```


1.5.12 高亮显示

首先指定哪些列需要高亮显示，然后在结果中得到高亮段。HighlightBuilder 定制在哪个域值的检索结果的关键字上增加高亮显示。

```
SearchRequestBuilder searchRequestBuilder =
        client.prepareSearch(indexname).setTypes(type);

//高亮
HighlightBuilder hiBuilder = new HighlightBuilder();
hiBuilder.preTags("<span style=\"color:red\">");
hiBuilder.postTags("</span>");

//高亮字段
hiBuilder.field("title");

searchRequestBuilder.highlighter(hiBuilder);
```

执行搜索，返回高亮显示文本的代码如下。

```
//返回搜索响应信息
SearchResponse response = searchRequestBuilder.execute().actionGet();
```

```
//获取搜索的文档结果
SearchHits searchHits = response.getHits();
for (SearchHit hit : searchHits) {
        //将文档中的每一个对象转换 json 串值
        String json = hit.getSourceAsString(); //搜索结果用 Gson 解析，解析要自己写

        //获取对应的高亮域
        Map<String, HighlightField> result = hit.highlightFields();
        //从设定的高亮域中取得指定域
        HighlightField titleField = result.get("title");
        //取得定义的高亮标签
        Text[] titleTexts = titleField.fragments();
        //为 title 串值增加自定义的高亮标签
        StringBuilder title = new StringBuilder();
        for (Text text : titleTexts) {
                title.append(text);
        }
        System.out.println("title highlighter " + title);
}
```


显示结果时，首先可以检查 highlightFields()，如果一列没有出现在那里，用 fields()代替。

```
public static String getTitle(HighlightField titleField,Map<String, Object> source){
        if(titleField= =null){   //如果没有高亮值，就显示原来的值
                return (String) source.get("title");
        }

        StringBuilder title = new StringBuilder();

        //取得定义的高亮标签
        Text[] titleTexts = titleField.fragments();
        //为 title 串值增加自定义的高亮标签
        for (Text text : titleTexts) {
                title.append(text);
        }

        return title.toString();
}
```

1.5.13　分页

设置两个参数：从第几个结果开始返回文档，以及最多返回多少个文档。返回一个结果总数，用于知道有多少页可以翻。

在 SearchRequestBuilder 中设置分页参数：

```
searchRequestBuilder.setFrom(0).setSize(60);
```

用于分页显示的代码如下。

```
int rows=10; //一页显示多少条搜索结果
int offset=0;   //开始行

//...
String keyWords = "hello"; //查询词
//设置查询关键词
QueryStringQueryBuilder qb = new QueryStringQueryBuilder(keyWords);
searchRequestBuilder.setQuery(qb);

//分页应用
searchRequestBuilder.setFrom(offset).setSize(rows);

//执行搜索,返回搜索响应信息
SearchResponse response = searchRequestBuilder.execute().actionGet();

//获取搜索的文档结果
SearchHits searchHits = response.getHits();
long totalHits = searchHits.getTotalHits();   //得到结果总数
for (SearchHit hit : searchHits) {
        System.out.println("hit " + hit);
}
```


1.5.14 通过聚合实现分组查询

使用 AggregationBuilders 实现搜索结果按某个不分词的列统计。基本用法：

```
AggregationBuilders.terms(aggName).field(filed)
```

使用代码如下。

```
TermsAggregationBuilder ab = AggregationBuilders.terms("version")
                .field("version"); //分类列

SearchResponse sr = client.prepareSearch(index).setQuery(qb)
                .addAggregation(ab) //增加分组查询到搜索请求
                .execute().actionGet();
```

可以限制返回结果数量：

```
ab.size(100);   //只显示前 100 个统计结果
```

得到每个 oid 名称，并且它的计数除以 1000。这里的数字 1000 可以更改。

```
Map<String, String> bucketsPathsMap = new HashMap<>();
bucketsPathsMap.put("count", "_count");
Script script = new Script("count / 1000");
```

```
BucketScriptPipelineAggregationBuilder bs = PipelineAggregatorBuilders
        .bucketScript("agg_values", bucketsPathsMap, script);

TermsAggregationBuilder oid = AggregationBuilders.terms("agg_oid")
        .field("oid").subAggregation(bs);

SearchResponse sr = client.prepareSearch(index)
        .setQuery(qb)
        .addAggregation(oid).execute().actionGet();
```

1.5.15　文本列的聚合

大多数字段都按默认情况索引，这使得它们可以搜索到。但排序、聚合和在脚本中访问字段值需要与搜索不同的访问模式。

搜索需要回答“哪些文档包含这个词”，而排序和聚合则需要回答一个不同的问题，即“这个文档的这个字段的值是什么”。

大多数字段可以使用索引时间，磁盘上的 doc_values 用于此数据访问模式，但文本字段不支持 doc_values。

文本字段使用称为 fielddata 查询内存数据结构。该数据结构是第一次将字段用于聚合、排序或脚本时构建的。它是通过从磁盘读取每个段的整个倒排索引构建的，反转词=>文档关系，并将结果存储在 JVM 堆内存中。

例如，在文本字段上使用 fielddata=true 的 mapping。

```
PUT my_index/_mapping/my_type
{
  "properties": {
    "labels": {
      "type":        "text",
      "fielddata": true
    }
  }
}
```

用该字段执行聚合：

```
public void prepare(final SearchRequestBuilder searchRequestBuilder) {
        TermsAggregationBuilder aggregation =
                        AggregationBuilders.terms("labels").field("labels");
        searchRequestBuilder.addAggregation(aggregation);
        searchRequestBuilder.setFrom(start);
        searchRequestBuilder.setSize(size);
}
```

使用过滤器实现：

```
//得到 QueryBuilder
String keyWords = "的";
String field = "body";
MatchQueryBuilder pqBody = QueryBuilders.matchPhraseQuery(field, keyWords);

field = "title";
MatchQueryBuilder pqTitle = QueryBuilders.matchPhraseQuery(field, keyWords);

QueryBuilder qb = QueryBuilders.boolQuery().should(pqBody)
        .should(pqTitle);

String index = "news"; //索引名

Client client = QueryTest.getClient();

SearchRequestBuilder requestBuilder = client.prepareSearch(index);

//在查询上增加过滤器
AndFilterBuilder andFilterBuilder = FilterBuilders.andFilter();
andFilterBuilder.add(FilterBuilders.termFilter("version", "3"));
requestBuilder.setQuery(QueryBuilders.filteredQuery(qb, andFilterBuilder));

//执行搜索
SearchResponse searchResponse = requestBuilder.execute().actionGet();

//解析搜索结果和响应
SearchHits hits = searchResponse.getHits();

long totalHits = hits.getTotalHits(); //得到结果总数
System.out.println("totalHits:" + totalHits);
```


1.5.16 遍历数据

使用游标查询数据。

```
SearchResponse searchResponse = esClient
        .prepareSearch(ElasticSearchClient.getIndexName())
        .setSize(5)
        //这个游标维持多长时间
        .setScroll(TimeValue.timeValueMinutes(8)).execute().actionGet();
System.out.println(searchResponse.getHits().getTotalHits());
searchResponse = esClient
        .prepareSearchScroll(searchResponse.getScrollId())
        .setScroll(TimeValue.timeValueMinutes(8)).execute().actionGet();
System.out.println(searchResponse.getHits().getTotalHits());
```

```
System.out.println(searchResponse.getHits().hits().length);
for (SearchHit hit : searchResponse.getHits()) {
        System.out.println(hit.getSourceAsString());
}
```

可以通过遍历数据的方式把一个索引中的数据复制到另外一个索引中。

1.5.17　索引文档

mapper 附件插件通过使用 Apache 文本提取库 Tika，让 Elasticsearch 可以索引一千多种格式（如 PPT、XLS、PDF）的文件附件。

实际上，插件在映射属性时添加附件类型，以便可以使用文件附件内容（编码为 base64）填充文档。

1.5.18　Percolate

Percolate 是一个根据文档倒过来查询的功能。可以把 Percolate 看成索引，然后搜索的反向操作。

Percolate 不是发送文档，索引它们，然后运行查询。Percolate 是发送一些查询，注册这些查询，然后发送文档并找出哪些查询匹配这个文档。

举一个例子，一个用户可以注册一个兴趣（查询）到所有的微博上。这个兴趣就是包含单词“雾霾”的文档。对于每一个微博，可以过滤这个微博到所有注册的用户查询，并找出哪些查询匹配了这个微博。

```
//要注册到过滤器的查询
QueryBuilder qb = termQuery("content", "amazing");

//索引这个查询 = 注册这个查询到过滤器
client.prepareIndex("_percolator", "myIndexName", "myDesignatedQueryName")
      .setSource(qb)
      .setRefresh(true) //当需要查询立即可用时，需要这个代码
      .execute().actionGet();
```

上面的代码以 myDesignatedQueryName 的名字索引一个词查询。

为了在注册的查询上检查一个文档，使用如下的代码。

```
//构建一个文档来检查过滤器
XContentBuilder docBuilder = XContentFactory.jsonBuilder().startObject();
docBuilder.field("doc").startObject(); //需要用它来指定文档
docBuilder.field("content", "This is amazing!");
docBuilder.endObject(); //doc 列结束
docBuilder.endObject(); //json 根对象结束
//过滤
PercolateResponse response =
            client.preparePercolate("myIndexName",
```

```
"myDocumentType").setSource(docBuilder).execute().actionGet();
//遍历结果
for(String result : response) {
    //处理结果，结果是过滤器中查询的名字
}
```

1.6 RESTClient

Elasticsearch 的 Java 客户端 API 依赖 httpclient、rest 及 Elasticsearch 相关的 jar 包。如果使用 maven，则需要添加依赖关系：

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>rest</artifactId>
    <version>5.3.0</version>
</dependency>
```


建立连接：

```
RestClient restClient = RestClient
            .builder(new HttpHost("127.0.0.1", 8310, "http"))
            .setMaxRetryTimeoutMillis(6 * 60 * 1000) //超时时间设为 6 分钟
            .build();
```

跟踪失败原因的例子：

```
public static final class FailureListenerEx extends FailureListener{
    public void onFailure(HttpHost host) {
        System.out.println("出错！"+host);
    }
}

public static void main(String[] args) {
    HttpHost hosts = new HttpHost("60.205.165.29", 9200, "http");

    FailureListener failureListener =  new FailureListenerEx();
    RestClientBuilder builder =
        RestClient.builder(hosts).setFailureListener(failureListener);
    builder.setMaxRetryTimeoutMillis(6 * 60 * 1000);
    builder.build();
}
```

通过认证访问 Es 集群的例子：

```
final CredentialsProvider credentialsProvider = new BasicCredentialsProvider();
credentialsProvider.setCredentials(AuthScope.ANY,
        new UsernamePasswordCredentials("elastic", "changeme"));
```

```
restClient = RestClient.builder(new HttpHost("192.168.8.33",9200,"http"))
        .setHttpClientConfigCallback(new RestClientBuilder.HttpClientConfigCallback() {
            public HttpAsyncClientBuilder customizeHttpClient(HttpAsyncClientBuilder
                                                    httpClientBuilder) {
                return httpClientBuilder
                        .setDefaultCredentialsProvider(credentialsProvider);
            }
        }).build();
```

REST 客户端分为低层和高层两部分。使用高层客户端察看文本分析结果的例子：

```
String method = "GET";
String endpoint = "/_analyze";
Map<String, String> params = new HashMap<String, String>();
params.put("analyzer","standard");
params.put("text","中华人民共和国");
Response response = restClient.performRequest(method,endpoint,params);
System.out.println(JSON.toJSONString(JSONObject.parse(EntityUtils.toString(response.getEntity())),Serializ
erFeature.PrettyFormat));
```


使用高层客户端删除文档的例子：

```
RestClient restClient = RestClientSingleton.getClient();
StringEntity query = new StringEntity(
        "{\"query\":{\"match\":{\"user\":\"user1\"}}}",
        ContentType.APPLICATION_JSON);

Response response = restClient.performRequest("GET",
        "/bulktest/_search",
        Collections.singletonMap("pretty", "true"), query);
System.out.println(EntityUtils.toString(response.getEntity()));
```

1.6.1　使用摄取

定义一个摄取管道小写的例子如下。

```
curl -XPUT localhost:9200/_ingest/pipeline/lowercase-example -d '{
   "processors" : [
      {
        "lowercase" : {
          "field": "message"
        }
      }
   ]
}'
```

可以使用_simulate 端点来模拟管道，而不需要实际创建管道和索引文档。

```
curl -XGET "localhost:9200/_ingest/pipeline/_simulate?pretty" -d '{
  "pipeline" : {
    "processors" : [
     {
       "lowercase" : {
         "field": "foo"
       }
     }
    ]
  },
  "docs" : [
    {
      "_source" : {
        "foo" : "This test CONTAINS also UPPER-CASE chars"
      }
    }
  ]
}'
```


这会得到：

```
{
  "_index" : "test",
  "_type" : "doc",
  "_id" : "1",
  "_version" : 1,
  "found" : true,
  "_source" : {
    "foo" : "this test contains also upper-case chars"
  }
}
```

为了即时转换文件，只需将管道 URL 参数添加到索引操作中。

```
curl -XPUT "localhost:9200/test/doc/1?pipeline=lowercase-example&pretty" -d '{
  "foo" : "This test CONTAINS also UPPER-CASE chars"
}'
```

看看真正发送到 Es 的_source 字段的值：

```
curl -XGET "localhost:9200/test/doc/1?pretty"
```

可以看到该文档已经转换了：

```
{
  "_index" : "test",
  "_type" : "doc",
  "_id" : "1",
  "_version" : 1,
```

```
    "found" : true,
    "_source" : {
      "foo" : "this test contains also upper-case chars"
    }
}
```

1.6.2　代码实现摄取

用代码来模拟管道：

```
String json = jsonBuilder().startObject().startObject("pipeline")
            .startArray("processors").startObject()
            .startObject("lowercase").endObject().endObject().endArray()
            .endObject().startArray("docs").startObject()
            .field("_index", "index").field("_type", "type")
            .field("_id", "id").startObject("_source").field("foo", "bar")
            .endObject().endObject().endArray().endObject().string();

Response response = client.performRequest("POST",
            "/_ingest/pipeline/_simulate", Collections.emptyMap(),
            new NStringEntity(json, ContentType.APPLICATION_JSON));
```


1.7　使用 Jest

Jest（https://github.com/searchbox-io/Jest）是另外一个 Java HTTP Rest 客户端。使用 Jest 不需要引用 Elasticsearch 本身的 jar 包。如果使用 maven，只需要添加依赖关系：

```
<dependency>
    <groupId>io.searchbox</groupId>
    <artifactId>jest</artifactId>
    <version>5.3.2</version>
</dependency>
```

使用它可以直接用 curl 命令的方式写代码。根据配置通过工厂构建新的 Jest 客户端：

```
JestClientFactory factory = new JestClientFactory();
factory.setHttpClientConfig(new HttpClientConfig
                            .Builder("http://localhost:9200")
                            .multiThreaded(true)
            //默认情况下，在给定的路由上创建不超过 2 个并发连接
            .defaultMaxTotalConnectionPerRoute(<CONCURRENCY_PER_ROUTE>)
            //总共不超过 20 个连接
            .maxTotalConnection(<CONCURRENCY_TOTAL>)
                            .build());
JestClient client = factory.getObject();
```

使用 Jest 获取 Es 集群信息。首先定义一个 Action 类：

```
import io.searchbox.action.AbstractAction;
import io.searchbox.action.GenericResultAbstractAction;

public class GetStartPage extends GenericResultAbstractAction {

    protected GetStartPage(Builder builder) {
        super(builder);
        setURI(buildURI());
    }

    protected String buildURI() {
        return super.buildURI() + "/?pretty";
    }

    @Override
    public String getRestMethodName() {
        return "GET";
    }

    public static class Builder extends AbstractAction.Builder<GetStartPage, Builder> {

        @Override
        public GetStartPage build() {
            return new GetStartPage(this);
        }
    }
}
```

然后使用它：

```
GetStartPage getStartPage = new GetStartPage.Builder().build();
try {
    JestResult execute = client.execute(getStartPage);
    System.out.println("result =" + execute.getJsonString());
} catch (IOException ex) {
    Logger.getLogger(Jest.class.getName()).log(Level.SEVERE, null, ex);
}
```

将数据索引到服务器的示例：

```
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder()
    .query(query)
    .fetchSource(new String[] {
      "oid",
      "events.activityoid",
      "events.worktime"
```

```
        }, null);
```

块索引：

```
    final Note note1 = new Note("mthomas", "Note1: do u see this - " +
            System.currentTimeMillis());
    Index index = new Index.Builder(note1).index(DIARY_INDEX_NAME)
            .type(NOTES_TYPE_NAME).build();
    jestClient.execute(index);
```

异步索引：

```
    final Note note2 = new Note("mthomas", "Note2: do u see this - " +
                                System.currentTimeMillis());
    index =
          new Index.Builder(note2).index(DIARY_INDEX_NAME)
          .type(NOTES_TYPE_NAME).build();
             jestClient.executeAsync(index, new JestResultHandler() {
                   public void failed(Exception ex) {
                   }

                   public void completed(JestResult result) {
                         note2.setId((String) result.getValue("_id"));
                         System.out.println("completed=&gt;&gt;" + note2);
                   }
             });
```

批量索引：

```
    final Note note3 = new Note("mthomas", "Note3: do u see this - " + System.currentTimeMillis());
    final Note note4 = new Note("mthomas", "Note4: do u see this - " + System.currentTimeMillis());
    Bulk bulk = new Bulk.Builder()
                .addAction(new
Index.Builder(note3).index(DIARY_INDEX_NAME).type(NOTES_TYPE_NAME). build())
                .addAction(new
Index.Builder(note4).index(DIARY_INDEX_NAME).type(NOTES_TYPE_NAME). build())
                .build();
    JestResult result = jestClient.execute(bulk);

    Thread.sleep(2000);

    System.out.println(result.toString());
```

查询数据：

```
    SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
    searchSourceBuilder.query(QueryBuilders.termQuery("note", "see"));

    Search search = new Search.Builder(searchSourceBuilder.toString())
```

```
            .addIndex(DIARY_INDEX_NAME)
            .addType(NOTES_TYPE_NAME).build();
System.out.println(searchSourceBuilder.toString());
JestResult result = jestClient.execute(search);
List notes = result.getSourceAsObjectList(Note.class);
for (Note note : notes) {
    System.out.println(note);
}
```

为了定义索引库结构需要用到 elasticsearch-X.Y.Z.jar 中的 XContentBuilder 类，实现代码如下。

```
//创建索引
jestClient.execute(new CreateIndex.Builder("my_index").build());

PutMapping putMapping = new PutMapping.Builder(
            "my_index",
            "my_type",
            "{ \"document\" : { \"properties\" : { \"message\" : {\"type\" : \"string\", \"store\" : \"yes\"} } } }")
            .build();

JestResult mappingResult = jestClient.execute(putMapping);
System.out.println(mappingResult.isSucceeded());

XContentBuilder docObject = jsonBuilder().startObject()
            .startObject("doc").field("message", "test terst").endObject()
            .endObject();

Index index = new Index.Builder(docObject.string()).index("my_index")
            .type("my_type").build();
JestResult jestResult = jestClient.execute(index);
System.out.println(jestResult.isSucceeded());
```


可以在构建客户端时配置基本的用户名和密码认证：

```
JestClientFactory factory = new JestClientFactory();
factory.setHttpClientConfig(
    new HttpClientConfig.Builder("http://localhost:9200")
        .defaultCredentials("global_user", "global_password")
        .build()
);
```

得到搜索结果中的文档编号：

```
List<Hit<Map,Void>> hits = client.execute(search).getHits(Map.class);
Hit hit = hits.get(0);
Map source = (Map)hit.source;
String id = (String)source.get(JestResult.ES_METADATA_ID);
```

1.8 Python 客户端

有些生产环境的集群通过跳板机才能接触到。为了方便在服务器端开发 Python 应用，可以采用 Micro（https://github.com/zyedidia/micro）这样的终端文本编辑器。

如果没有 dnf 安装工具软件，可以直接安装 Micro 的预编译版本：

```
#wget https://github.com/zyedidia/micro/releases/download/nightly/micro-1.2.1-10-linux64.tar.gz
#tar -xf ./micro-1.2.1-10-linux64.tar.gz
```

可以使用它编辑代码文件：

```
./micro    test.py
```

保存文件后，按 Ctrl+Q 组合键退出。

elasticsearch-py（https://github.com/elastic/elasticsearch-py）是 Elasticsearch 的官方底层 Python 客户端。Elasticsearch DSL（https://github.com/elastic/elasticsearch-dsl-py）是高层客户端。

先介绍 elasticsearch-py 的使用。用 pip 安装 elasticsearch 包：

```
pip install elasticsearch
```

简单的用法如下。

```
>>> from datetime import datetime
>>> from elasticsearch import Elasticsearch

#默认情况下连接到 localhost:9200
>>> es = Elasticsearch()

#在 elasticsearch 中创建一个索引，忽略状态码 400（索引已经存在）
>>> es.indices.create(index='my-index', ignore=400)
{u'acknowledged': True}

#会序列化 datetime
>>> es.index(index="my-index", doc_type="test-type", id=42, body={"any": "data", "timestamp": datetime.now()})
{u'_id': u'42', u'_index': u'my-index', u'_type': u'test-type', u'_version': 1, u'ok': True}

#但没有反序列化
>>> es.get(index="my-index", doc_type="test-type", id=42)['_source']
{u'any': u'data', u'timestamp': u'2016-05-12T19:45:31.804229'}
```

然后介绍 Elasticsearch DSL 的使用，直接写成一个 dict 的典型的搜索请求如下。

```
from elasticsearch import Elasticsearch
client = Elasticsearch()

response = client.search(
```

```
        index="my-index",
        body={
          "query": {
            "bool": {
              "must": [{"match": {"title": "python"}}],
              "must_not": [{"match": {"description": "beta"}}],
              "filter": [{"term": {"category": "search"}}]
            }
          },
          "aggs" : {
            "per_tag": {
              "terms": {"field": "tags"},
              "aggs": {
                "max_lines": {"max": {"field": "lines"}}
              }
            }
          }
        }
    )

    for hit in response['hits']['hits']:
        print(hit['_score'], hit['_source']['title'])

    for tag in response['aggregations']['per_tag']['buckets']:
        print(tag['key'], tag['max_lines']['value'])
```

这种方法的问题是它非常冗长，容易出现语法错误，如错误的嵌套、难以修改（如添加另一个过滤器）。

用 Python DSL 重写示例如下。

```
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search

client = Elasticsearch()

#将术语查询放在布尔查询的过滤器上下文中
s = Search(using=client, index="my-index") \
    .filter("term", category="search") \
    .query("match", title="python")   \
    .exclude("match", description="beta")

s.aggs.bucket('per_tag', 'terms', field='tags') \
    .metric('max_lines', 'max', field='lines')

response = s.execute()
```

```
for hit in response:
    print(hit.meta.score, hit.title)

for tag in response.aggregations.per_tag.buckets:
    print(tag.key, tag.max_lines.value)
```

用一个简单的 Python 类，代表博客系统中的一篇文章：

```
from datetime import datetime
from elasticsearch_dsl import DocType, Date, Integer, Keyword, Text
from elasticsearch_dsl.connections import connections

#定义一个默认的 Elasticsearch 客户端
connections.create_connection(hosts=['localhost'])

class Article(DocType):
    title = Text(analyzer='snowball', fields={'raw': Keyword()})
    body = Text(analyzer='snowball')
    tags = Keyword()
    published_from = Date()
    lines = Integer()

    class Meta:
        index = 'blog'

    def save(self, ** kwargs):
        self.lines = len(self.body.split())
        return super(Article, self).save(** kwargs)

    def is_published(self):
        return datetime.now() > self.published_from

#在 elasticsearch 中创建映射
Article.init()

#创建并保存文章
article = Article(meta={'id': 42}, title='Hello world!', tags=['test'])
article.body = ''' looong text '''
article.published_from = datetime.now()
article.save()

article = Article.get(id=42)
print(article.is_published())

#显示群集运行状况
print(connections.get_connection().cluster.health())
```

接下来介绍使用摄取处理器插件。首先安装 attachment 插件：

```
# sudo bin/elasticsearch-plugin install ingest-attachment
```

然后重新启动 Es，让它生效。

在 Python 客户端中创建一个管道，然后使用摄取 Attachment 处理器插件，代码如下。

```
from elasticsearch import Elasticsearch
es = Elasticsearch()
body = {
        "description" : "Extract attachment information",
         "processors" : [
          {
                "attachment" : {
                "field" : "data"
                    }
           }
         ]
       }
es.index(index='_ingest', doc_type='pipeline', id='attachment', body=body)
```


1.9 Scala 客户端

Elastic4s（https://github.com/sksamuel/elastic4s）是 Elasticsearch 的一个简洁的 Scala 客户端。客户端可以通过选择 elastic4s-http 或 elastic4s-tcp 子模块中的任何一个来使用 HTTP 和 TCP。

当然也可以在 Scala 中使用官方的 Elasticsearch Java 客户端，但由于 Java 的语法更为冗长，并且它不支持 Scala 核心库中的类，也不支持 Scala 的习惯用法。

Elastic4s 的 DSL 允许以编程方式构建请求，并在编译时显示句法和语义错误，且使用标准 Scala Future 使你能够轻松地集成到异步工作流中。DSL 的目标是请求以类似构建器的方式编写的，与 Java API 或 Rest API 大致相似。每个请求都是一个不可变的对象，因此可以创建请求并安全地重用它们，或者为了派生请求进一步复制它们。因为每个请求是强类型的，所以 IDE 或编辑器可以使用类型信息来显示哪些操作可用于当前的请求类型。

Elastic4s 支持 Scala 集合，所以不必从 Scala 域类到 Java 集合中进行烦琐的转换。它还允许使用类型类直接索引和读取类，因此不必手动设置字段或 json 文档。这些类型类可以使用如下任意的 json 库生成：存在的模块包括 Jackson、Circe、Json4s、PlayJson 和 Spray Json。客户端还使用标准的 Scala 持续时间来避免使用字符串或原语的持续时间。

Elastic4s 发布了 Scala 2.11 和 Scala 2.12。对 Scala 2.10 的支持已经从 5.0.x 发布中删除了。对于与早期版本的 Elasticsearch 兼容的版本，可搜索 maven 中心。

从版本 5.0.0 开始，底层的 Elasticsearch TCP Java 客户端依赖于 Netty、Lucene 等。Elastic4s TCP 客户端为你带来依赖关系，但如果有任何错误，就需要将其添加到自己的构建中。

另一个问题是它使用 Netty 4.1。然而，一些流行的项目，如 Spark 和 Play 目前使用 4.0

版，两个版本之间有一个突破性的变化。因此，如果使用 Elastic4s TCP（甚至只是 Elasticsearch Java TCP 客户端），尝试将它与 Play 或 Spark 一起使用，则都会得到 NoSuchMethodException 错误。

Elasticsearch（在 JVM 上）有两个接口。一个是端口 9200 上可用的常规 HTTP 接口（默认情况下），另一个是端口 9300 上的 TCP 接口（默认情况下）。历史上，Elasticsearch 提供的 Java API 一直是基于 TCP 的，其基本原理是将编组请求保存到 json 中，并且具有集群感知能力，所以可以将请求路由到正确的节点。因此，Elastic4s 也是基于 TCP 的，因为它将请求委托给底层的 Java 客户端。

从 Elastic4s 5.2.x 开始，添加了一个依赖于 Java REST 客户端进行连接管理的新 HTTP 客户端，但仍然使用大家熟悉的 Elastic4s DSL 来构建查询。

在 5.2 和 5.3 版本链上经过广泛测试后，从版本 5.4.x 起，HTTP 客户端可以在生产环境下使用。

根据所使用的客户端，需要在构建中添加一个 elastic-http 或 elastic-tcp 依赖关系。

为了搭建 Scala 开发环境，首先安装 Scala，然后安装 Eclipse 的 Scala 插件。可以从 http://scala-ide.org/download/prev-stable.html 下载 Scala 插件。

首先需要添加一个依赖关系：elastic4s-http 或 elastic4s-tcp。

基本用法是创建客户端的实例，然后使用要执行的请求调用 execute 方法。execute 方法是异步的，并返回一个标准的 Scala Future，其中 T 是适用于请求类型的响应类型。例如，搜索请求将返回包含搜索结果的 SearchResponse 类型的响应。

要创建 HTTP 客户端的实例，可以使用 HttpClient 协同对象方法。要创建 TCP 客户端的实例，可以使用 TcpClient 协同对象方法。对于任意一个客户端，请求是相同的，但响应类可能会稍微变化，因为 HTTP 响应类建模返回的是 json，而 TCP 响应类则包装 Java 客户端类。

请求使用 Elastic4s DSL 创建。例如，要创建搜索请求，可以执行以下操作：

```
search("index" / "type").query("findthistext")
```

DSL 方法位于需要导入或扩展的 ElasticDsl trait 中。尽管无论使用 HTTP 还是 TCP 客户端，语法都是一样的，还是必须根据使用的客户端导入相应的 trait（com.sksamuel.elastic4s.ElasticDSL for TCP 或 com.sksamuel.elastic4s.http.ElasticDSL for HTTP）。

示例 SBT 设置：

```
//major.minor 与 elasticsearch 版本同步
val elastic4sVersion = "x.x.x"
libraryDependencies ++= Seq(
  "com.sksamuel.elastic4s" %% "elastic4s-core" % elastic4sVersion,
  //用于 TCP 客户端
  "com.sksamuel.elastic4s" %% "elastic4s-tcp" % elastic4sVersion,

  //用于 HTTP 客户端
  "com.sksamuel.elastic4s" %% "elastic4s-http" % elastic4sVersion,

  //如果要使用响应式流
```

```
    "com.sksamuel.elastic4s" %% "elastic4s-streams" % elastic4sVersion,

    //测试
    "com.sksamuel.elastic4s" %% "elastic4s-testkit" % elastic4sVersion % "test",
    "com.sksamuel.elastic4s" %% "elastic4s-embedded" % elastic4sVersion % "test"
)
```

使用客户端连接到一个节点，创建一个索引并索引一个列文档，然后使用简单的文本查询来搜索该文档。

```
import com.sksamuel.elastic4s.{ElasticsearchClientUri, TcpClient}
import com.sksamuel.elastic4s.searches.RichSearchResponse
import org.elasticsearch.action.support.WriteRequest.RefreshPolicy
import org.elasticsearch.common.settings.Settings

//引入 json 库 circe
import com.sksamuel.elastic4s.circe._
import io.circe.generic.auto._

case class Artist(name: String)

object ArtistIndex extends App {

    //产生嵌入式节点进行测试
    val localNode = LocalNode(clusterName, homePath.toAbsolutePath.toString)

    //连接到本地节点以获取客户端对象
    //在实际代码中，必须使用 HttpClient 或 TcpClient 创建客户端
    val client = localNode.elastic4sclient()

    //必须导入 DSL
    import com.sksamuel.elastic4s.ElasticDsl._

  //接下来预先创建一个索引，准备接收文档
  //await 是一个帮助方法，让这个操作同步而不是异步
  //通常会避免在一个真正的程序中这样做，因为这样会阻止调用线程，但在测试时很有用
  client.execute {
    createIndex("bands").mappings(
        mapping("artist") as(
            textField("name")
        )
    )
  }.await

  //接下来索引一个文档，请注意，可以直接传入样例类
  //而 Elastic4s 将会使用之前导入的 circe 编组器来编组这个类
```

```
    //这里指定的刷新策略是希望此文档立即刷新到磁盘
    client.execute {
        indexInto("bands" / "artists") doc Artist("Coldplay") refresh(RefreshPolicy.IMMEDIATE)
    }.await

    //现在可以搜索刚刚索引的文档
    val resp = client.execute {
       search("bands" / "artists") query "coldplay"
    }.await

    println("---- Search Hit Parsed ----")
    resp.to[Artist].foreach(println)

    //打印完整的回复
    import io.circe.Json
    import io.circe.parser._
    println("---- Response as JSON ----")
    println(decode[Json](resp.original.toString).right.get.spaces2)

    client.close()
}
```


1.10　PHP 客户端

Elastica（https://github.com/ruflin/Elastica/）是一个 PHP 客户端。使用 Elastica 通过遍历索引库的方式找出一个 ID 的代码如下。

```
<?php
//创建一个 Elastica 客户端连接到 Es 服务器
$client = new \Elastica\Client (array('host'=>'localhost','port'=>'9200'));
//将客户端添加到搜索对象
$search = new \Elastica\Search($client);
//过滤类型
//$search->addTypes(['some_type']);
//创建 null 查询，因为想要所有的
$query = new \Elastica\Query(null);
//选择想要的列
$query->setStoredFields(['id','type']);
//将页面大小限制为一个结果，因为想要在每个记录上使用 getId()和 getType()
$query->setSize(1);
//将查询添加到搜索对象
$search->setQuery($query);
//返回 1 分钟滚动超时的滚动迭代器（默认值）
$scrollIterator = $search->scroll('1m');
//执行搜索
```

```
$scrollIterator->rewind();

while ($scrollIterator->valid()) {
    //当 foreach 调用 next 函数时，scroll 会获取记录
    foreach ($scrollIterator as $page) {
        $document = $page->current();

        echo $document->getId() . " " . $document->getType() . " \n";
    }
}
?>
```

高亮的代码：

```
$matchQuery = new MatchPhrase('phrase', 'ruflin');
$query = new Query($matchQuery);
$query->setHighlight([
    'pre_tags' => ['<em class="highlight">'],
    'post_tags' => ['</em>'],
    'fields' => [
        'phrase' => [
            'fragment_size' => 200,
            'number_of_fragments' => 1,
        ],
    ],
]);
$type = $index->getType('helloworld');
$resultSet = $type->search($query);
foreach ($resultSet as $result) {
    $highlight = $result->getHighlights();//返回结果'My name is <em class="highlight">ruflin</em>'
}
```


1.11 SQL 支持

目前 Elasticsearch 公司的主要方向还在搜索上，并不自带对 SQL 的支持。

可以通过 Presto 或 Drill 的插件来实现 SQL 查询支持。Presto Elasticsearch Connector 的基于插件的方案难以充分发挥 Elasticsearch 的性能优势。因此，基于 Calcite 让 Elasticsearch 支持 SQL 的项目开发很活跃。

Apache Calcite 是一个用来建立数据库和数据管理系统的开源框架。它包括一个 SQL 解析器，一个在关系代数上构建表达式的 API，以及一个查询计划引擎。作为一个框架，Calcite 不存储自己的数据或元数据，而是通过插件的方式允许访问外部数据和元数据。

对于一个查询操作会经历如下流程：Calcite 会解析 SQL 并将其转换成逻辑执行计划，其间会根据当前连接中 Schema 定义的信息初始化每一个 Schema，然后根据查询中指定的 Schema 调用对应的 getTableMap 函数获取元数据，根据这个信息判断查询中出现的表名、

字段名是否正确以及检查 SQL 语法是否符合规范。然后再使用 Calcite 内部默认的实现生成物理执行计划，这个查询计划是树状结构的，最底层的节点是 ScanTable 操作（类似于 SQL 执行过程中首先执行 FROM 子句），对每一个表获取该表的数据，这时使用的算子为默认的 EnumerableTableAccessRel，再去调用具体 ScannableTable 的 scan 方法获取表的数据。完成后根据原始表的数据进行上层的 JOIN、FILTER、GROUP BY、SORT、LIMIT，甚至子查询等操作。

首先用命令行工具 sqlline 查询 csv 文件，然后使用代码测试。

下载 Calcite：

```
$ git clone https://github.com/apache/calcite.git
```

构建 Calcite：

```
$ cd calcite
$ mvn install -DskipTests -Dcheckstyle.skip=true
$ cd example/csv
```

现在使用 sqlline 连接到 Calcite。sqlline 是一个包含在这个项目中的 SQL shell。

```
$ ./sqlline
sqlline> !connect jdbc:calcite:model=target/test-classes/model.json admin admin
```

example.json 文件内容如下。

```
{
  version: '1.0',
  defaultSchema: 'STREAM',
  schemas: [
    {
      name: 'SS',
      tables: [
        {
          name: 'ORDERS',
          type: 'custom',
          factory: 'org.apache.calcite.adapter.csv.CsvStreamTableFactory',
          stream: {
            stream: true
          },
          operand: {
            file: 'sales/SORDERS.csv',
            flavor: "scannable"
          }
        }
      ]
    }
  ]
}
```

schemas 定义了一些 schema，也就是数据库。每一个 schema 指定了 name、type（可以分为 Map Schema、Custom Schema 和 JDBC Schema）。

这里的 Custom Schema 意味着只需指定 factory 和可选的 operand 参数（map 结构），schema 都是通过指定的 factory 类创建出来的（它需要实现 org.apache.calcite.schema.SchemaFactory 接口），具体这个 schema 下面有哪些表可通过 schema 的 name 和 operand 变量决定是否生成。

SORDERS.csv 文件内容如下。

```
PRODUCTID:int,ORDERID:int,UNITS:int
3,4,5
2,5,12
2,1,6
```

查询测试代码如下。

```
Class.forName("org.apache.calcite.jdbc.Driver");
Properties info = new Properties();
info.setProperty("lex", "JAVA");
Connection connection =
    DriverManager.getConnection("jdbc:calcite:model="
      + "d:/Downloads/calcite-master/example/csv/target/test-classes/example.json",info);
CalciteConnection calciteConnection =
    connection.unwrap(CalciteConnection.class);
Statement statement = connection.createStatement();
ResultSet resultSet =
        statement.executeQuery("select STREAM * from SS.ORDERS where SS.ORDERS.UNITS > 5");
final StringBuilder buf = new StringBuilder();
while (resultSet.next()) {
  int n = resultSet.getMetaData().getColumnCount();
  for (int i = 1; i <= n; i++) {
    buf.append(resultSet.getMetaData().getColumnLabel(i))
        .append("=")
        .append(resultSet.getObject(i));
  }
  System.out.println(buf.toString());
  buf.setLength(0);
}
resultSet.close();
statement.close();
connection.close();
```

这里的 STREAM 关键字是流式 SQL 的主要扩展。它告诉系统，你感兴趣的是输入的订单，而不是现有的。

https://github.com/apache/calcite 中包含了 Elasticsearch 插件，一个模型定义如下。

```
{
```

```
 "version": "1.0",
 "defaultSchema": "elasticsearch",
 "schemas": [
   {
     "type": "custom",
     "name": "elasticsearch_raw",
     "factory": "org.apache.calcite.adapter.elasticsearch.ElasticsearchSchemaFactory",
     "operand": {
       "coordinates": "{'127.0.0.1': 9300}",
       "userConfig": "{'bulk.flush.max.actions': 10, 'bulk.flush.max.size.mb': 1}",
       "index": "usa"
     }
   },
   {
     "name": "elasticsearch",
     "tables": [
       {
         "name": "ZIPS",
         "type": "view",
         "sql": [
           "select cast(_MAP['city'] AS varchar(20)) AS \"city\",\n",
           " cast(_MAP['loc'][0] AS float) AS \"longitude\",\n",
           " cast(_MAP['loc'][1] AS float) AS \"latitude\",\n",
           " cast(_MAP['pop'] AS integer) AS \"pop\",\n",
           " cast(_MAP['state'] AS varchar(2)) AS \"state\",\n",
           " cast(_MAP['id'] AS varchar(5)) AS \"id\"\n",
           "from \"elasticsearch_raw\".\"zips\""
         ]
       }
     ]
   }
 ]
}
```

查询代码如下。

```
Class.forName("org.apache.calcite.jdbc.Driver");
Properties info = new Properties();
info.setProperty("lex", "JAVA");
Connection connection =
    DriverManager.getConnection("jdbc:calcite:model="
      + "d:/Downloads/calcite-master/example/elasticsearch-zips-model.json",info);
CalciteConnection calciteConnection =
    connection.unwrap(CalciteConnection.class);
Statement statement = connection.createStatement();
ResultSet resultSet =
        statement.executeQuery("select * from zips where \"pop\" in (20012, 15590)");
```

1.12 本章小结

Cutting 在 1999 年写 Lucene 以前，在苹果公司用 C++开发过搜索引擎。Lucene 是他写的第一个 Java 软件。在后来的 10 多年里，Lucene 越来越流行，成为开源组织 Apache 基金会的项目，并在维基百科网站等项目中得到广泛使用。Cutting 后来开发 MapReduce 的 Java 版本 Hadoop 也同样成功。Cutting 因此进入 Apache 基金董事会，并在 2010 年成为董事会主席。

可以通过 Mapping 定义索引结构，然后填充数据到索引，最后再做搜索部分。

Shay Banon 在开发 Elasticsearch 的前身 Compass 时，意识到需要开发一个分布式的搜索解决方案。Elasticsearch 的第一个版本于 2009 年发布。Elasticsearch 公司成立于 2012 年。这家公司专门做开源分布式搜索引擎，是一个分布式的组织。该公司有两个地点：位于美国加利福尼亚的洛杉矶和位于荷兰的阿姆斯特丹。该公司还有小的办公室：美国亚利桑那州的凤凰城、巴黎、布拉格、奥斯汀、波士顿、巴塞罗那、柏林和罗马尼亚。

建立一个国际化的商业组织是复杂和昂贵的。Elasticsearch 公司仅仅创立 6 个月后，就获得了 1000 万美元的投资。2012 年，Elasticsearch 公司创始人 Steven Schuurman 和 Shay Banon 宣布，他们从风投公司 Benchmark Capital、Rod Johnson 和 Data Collective 筹得第一轮融资。在此之前，Schuurman 与 Johnson 共同创建了 SpringSource 公司，并且这家公司在 2009 年以 4 亿多美元卖给了 VMware。SpringSource 的几个创始人在创建这家公司之前从来没见过面。在将来，利用 P2P 网络，可以降低分布式搜索系统的运营成本。

Elasticsearch 的引擎是一个运行时环境，使人们能够实时地分析和处理大量数据。它“不要求你必须是一位数据科学家才能把它用好”，这也是该产品的一个主要亮点。Elasticsearch 存在的理由是把大数据的复杂性化解成像苹果产品那般简单。

Mailgun 收发大量电子邮件，跟踪和存储每封邮件发生的每个事件。每个月会新增数十亿事件，需要展示给我们的客户，为方便他们很容易地分析数据，就是全文搜索。利用 Elasticsearch 和 Logstash 技术可以满足这个需求。

Elasticsearch 2.0 以前的版本可以使用 Elasticsearch river 同步数据库中的数据到 Elasticsearch。因为 Elasticsearch river 在 Elasticsearch 进程内部运行，运行 river 需要额外的内存，更多的套接字，文件描述符等。这样可能导致集群不稳定。Elasticsearch river 被废弃后，一些 Elasticsearch river 后来实现为 Logstash 插件。

从 5.x 版本开始，Elasticsearch 的服务器端代码开始成熟。但客户端代码仍然在发展和调整之中。Java API 与 Es 服务器在端口 9300 上打交道，而 RESTful 的 HTTP 客户端 Jest 使用端口 9200。Jest 提供自己的 Java API，还可以使用 ES Java API 来构建查询，然后提交给 RESTful 端点。

从 5.0 版本开始，Elasticsearch 推出了自己的 Rest Client。

除了使用 Java API，还可以使用.Net 开发 Elasticsearch 搜索客户端。

2014 年出现了 Apache Calcite 项目。之后出现了数个支持 Elasticsearch 的 SQL 查询的插件版本。在将来，自然语言查询文本大数据的技术会逐渐走向成熟。

curl 是 Linux 下的一个 http 命令行工具。通过 curl 发送命令和 Elasticsearch 节点交互。

开 发 插 件

为了支持中文搜索，需要开发中文分词插件。

2.1 搜索中文

实现一个继承 AbstractIndexAnalyzerProvider 的类，提供自己的分析器实现，然后在配置文件的 type 值中声明类的全名。

2.1.1 中文分词原理

首先执行原子切分，处理中文串中的数字或英文字符串，然后执行二元概率切分，代码如下。

```
//原子切分
fstSeg.seg(sentence,g);
//二元分词
GraphFactory.seg(sentence, g);
```

测试分词：

```
String sentence = "巨星和苯磺隆为高效内吸药剂，一般施药后一周左右杂草开始枯黄死亡";
Segmenter seg = new Segmenter();
List<WordTokenInf> words = seg.split(sentence);
for (WordTokenInf word : words) {
        System.out.print(word.termText + " ");
}
```

为了更准确地搜索，可以标注词性。为了方便指明词的词性，可以给每个词性编码。例如，根据英文缩写，把“形容词”编码成 a、名词编码成 n、动词编码成 v 等。表 2-1 是完整的词性编码表。

表 2-1　词性编码表

代　码	名　称	举　例
a	形容词	最/d　大/a　的/u
ad	副形词	一定/d　能够/v　顺利/ad　实现/v　。/w
ag	形语素	喜/v　煞/ag　人/n
an	名形词	人民/n　的/u　根本/a　利益/n　和/c　国家/n　的/u　安稳/an　。/w
b	区别词	副/b　书记/n　王/nr　思齐/nr
c	连词	全军/n　和/c　武警/n　先进/a　典型/n　代表/n
d	副词	两侧/f　台柱/n　上/f　分别/d　雄踞/v　着/u
dg	副语素	用/v　不/d　甚/dg　流利/a　的/u　中文/nz　主持/v　节目/n　。/w
e	叹词	嗬/e　！/w
f	方位词	从/p　一/m　大/a　堆/q　档案/n　中/f　发现/v　了/u
g	语素	例如，dg 或 ag
h	前接成分	目前/t　各种/r　非/h　合作制/n　的/u　农产品/n
i	成语	提高/v　农民/n　讨价还价/i　的/u　能力/n　。/w
j	简称略语	民主/ad　选举/v　村委会/j　的/u　工作/vn
k	后接成分	权责/n　明确/a　的/u　逐级/d　授权/v　制/k
l	习用语	是/v　建立/v　社会主义/n　市场经济/n　体制/n　的/u　重要/a　组成部分/l　。/w
m	数词	科学技术/n　是/v　第一/m　生产力/n
n	名词	希望/v　双方/n　在/p　市政/n　规划/vn
ng	名语素	就此/d　分析/v　时/Ng　认为/v
nr	人名	建设部/nt　部长/n　侯/nr　捷/nr
ns	地名	北京/ns　经济/n　运行/vn　态势/n　喜人/a
nt	机构团体	[冶金/n　工业部/n　洛阳/ns　耐火材料/l　研究院/n]nt
nx	字母专名	ATM/nx　交换机/n
nz	其他专名	德士古/nz　公司/n
o	拟声词	汩汩/o　地/u　流/v　出来/v
p	介词	往/p　基层/n　跑/v　。/w
q	量词	不止/v　一/m　次/q　地/u　听到/v　，/w
r	代词	有些/r　部门/n
s	处所词	移居/v　海外/s　。/w
t	时间词	当前/t　经济/n　社会/n　情况/n
tg	时语素	秋/Tg　冬/tg　连/d　旱/a
u	助词	工作/vn　的/u　政策/n
ud	结构助词	有/v　心/n　栽/v　得/ud　梧桐树/n
ug	时态助词	你/r　想/v　过/ug　没有/v
uj	结构助词的	迈向/v　充满/v　希望/n　的/uj　新/a　世纪/n
ul	时态助词了	完成/v　了/ ul
uv	结构助词地	满怀信心/l　地/uv　开创/v　新/a　的/u　业绩/n
uz	时态助词着	眼看/v　着/uz
v	动词	举行/v　老/a　干部/n　迎春/vn　团拜会/n

续表

代　码	名　称	举　例
vd	副动词	强调/vd　指出/v
vg	动语素	做好/v　尊/vg　干/j　爱/v　兵/n　工作/vn
vn	名动词	股份制/n　这种/r　企业/n　组织/vn　形式/n　，/w
w	标点符号	生产/v　的/u　5G/nx　、/w　8G/nx　型/k　燃气/n　热水器/n
x	非语素字	生产/v　的/u　5G/nx　、/w　8G/nx　型/k　燃气/n　热水器/n
y	语气词	已经/d　30/m　多/m　年/q　了/y　。/w
z	状态词	势头/n　依然/z　强劲/a　；/w

例如，“不/d　忘/v　群众/n　疾苦/n　温暖/v　送/v　进/v　万/m　家/q”，可以使用隐马尔可夫模型（Hidden Markov Model，HMM）实现词性标注。

2.1.2　中文分词插件原理

Lucene 处理同义词时需要把同义词表示为一个图的形式。例如，当应用同义词：

```
domain name system → dns
```

到所处理的文本：

```
domain name system is fragile
```

使用属性 PositionIncrementAttribute 和 PositionLengthAttribute 编码的词图，如图 2-1 所示。

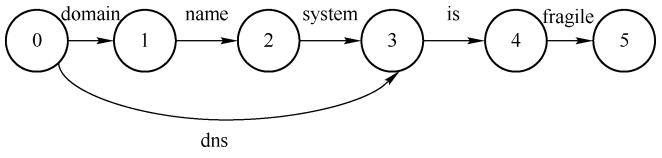

图 2-1　增加同义词 dns 的词图

然而，一旦将此文档添加到 Lucene 索引中，一些图结构就丢失了。因为 Lucene 忽略了 PositionLengthAttribute 属性，该属性会规定一个给定的符号何时终止。这样导致词图平坦化成图 2-2 所示。

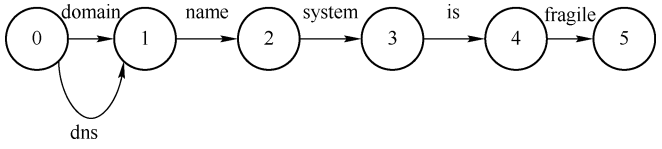

图 2-2　平坦化的词图

这样，查询“dns is fragile”便无法匹配它应该匹配的文档。同样，查“dns name system”不应该匹配到这个文档，实际却会匹配上。

更糟糕的是，如果同义词插入了多个符号，如将上述折叠规则反转为一个扩展规则：

```
dns → domain name system
```

然后分析如下文本：

```
dns is fragile
```

那么即使是由 SynonymFilter 生成的符号，在索引之前就已经被平坦化了！这是 Lucene 面临的同义词处理方面的问题。

Elasticsearch 5.2.0 中包含的 Lucene 6.4.0 有了让人满意的解决办法，可以进行同义词查询（只要是在搜索时间而不是索引时间应用同义词）。

第一个大的改变是增加了 SynonymGraphFilter 替换过时的 SynonymFilter。这个新的同义词过滤器可以在任何情况下生成正确的图表示，而不管你输入的是单个符号还是多个符号。此外，还有一个新的 FlattenGraphFilter，可以压缩图符号流以便用于索引。如果确实需要旧的 SynonymFilter 一样的行为，那么可以在 SynonymGraphFilter 后接一个过滤器 FlattenGraphFilter。但要记住 FlattenGraphFilter 有时会丢掉一些图结构。

改进的第二个方面是提高了查询解析器的效率。首先，旧的经典查询解析器在空格时会终止解析预切分的输入查询文本，记住调用 setSplitOnWhitespace(false)，因为空格切分是默认的设置以便向后兼容。这样就让查询时的解析器可以看见多个符号而不是把每个符号分开来处理。这些对 QueryBuilder 中复杂逻辑的简化是一个重要的引导。

第三个方面是检查查询分析器何时产生图表示，并生成合适的查询。现在查询分析器（也就是基类 QueryBuilder）可以监控 PositionLengthAttribute，当符号的 PositionLength Attribute 值大于 1 时计算图所有的路径。在 Elasticsearch 中，match_query、query_string 和 simple_query_string 查询都可以正确处理词图。

上述的改进可以让你在查询时使用多符号同义词，并取得精确的结果。例如，如果查询：

```
dns is fragile
```

如果没有引号，那么过滤器 SynonymGraphFilter 可以将 dns 扩展到 domain name system。然后，查询分析器将分析查询建立整个图表示，并计算整个图中的不同路径：

```
dns is fragile
domain name system is fragile
```

它会分别分析上述两个字符串，产生两个子查询，并使用 SHOULD 子句将它们组合在 BooleanQuery 中。如果原始查询拥有两个引号，将使用 PhraseQuery 将两者组合，产生出确定的答案。

在 Lucene 6.5.0 中，对查询器 QueryBuilder 的优化需要分析查询的图表示（关节点），以生成更有效的二值查询器，避免可能的路径组合维度过大。这种优化主要难点包括如何存储图查询、如何将二值处理器应用到同义词扩展、属性 minShouldMatch 如何工作等。例如，如果用户没有加括号，但插入多符号查询，该使用短语查询是否还是默认的查询解析器操作符？希望图查询的实践经验对这些问题提供一些启示。

除了同义词外，Lucene 还有更多的生成图表示的图过滤器。在 Lucene6.5.0 版本中，WordDelimiterFilter 将会被替换成 WordDelimiterGraphFilter。针对日语的处理器 Japanese

Tokenizer 基于 Kuromoji 日语形态分析器，已经可以生成词图来表示整个单词和可能的子单词。还有其他的词过滤器，如 Shingles、Ngrams 等。它们都应该生成图输出，但是还没有修订。

为了处理多符号同义词，必须在查询时使用同义词处理器，而不是在索引时使用，因为 Lucene 的索引还不能存储词图表示。和索引时同义词处理相比，查询时的同义词处理需要更多的 CPU 和 I/O 工作量，因为需要访问更多的术语来回答问题，但索引库小多了。查询时的同义词处理也更灵活，因为当改变同义词时，不需要再次建立索引。

另外一个挑战是 SynonymGraphFilter，它在生成正确的图表示时，不能处理图表示，这意味着在使用 WordDelimiterGraphFilter 或 JapaneseTokenizer 后不能使用 SynonymGraphFilter，而且希望同义词会匹配输入图的片段。

2.1.3　开发中文分词插件

写个支持 Lucene 6.4.2 的分词器。用 org.apache.lucene.analysis.BaseTokenStreamTestCase 写个测试类。

项目中除了基本的 lucene-core-6.4.1.jar、lucene-codecs-6.4.1.jar、junit-4.10.jar，还需要添加 randomizedtesting-runner-2.4.0.jar、lucene-test-framework-6.4.1.jar 这两个.jar。

如果使用 maven，则需要添加依赖关系：

```
<dependency>
        <groupId>org.apache.lucene</groupId>
                <artifactId>lucene-test-framework</artifactId>
        <version>6.4.1</version>
        <scope>test</scope>
</dependency>
```

测试类需要生成随机值，用所有可能的数据测试用户的代码，以便将来不会出现任何类型的数据失败。RandomizedTesting 是一个随机测试基础设施。

RandomizedTest.randomInt()方法得到随机整型值。一个使用 RandomizedTest 的例子：

```
public class TestUsingRandomness extends RandomizedTest {

   @Test
   public void expectNoException() {
      String [] words = {"oh", "my", "this", "is", "bad."};

      //这将从上面的数组中选出一个随机词
      System.out.println(words[Math.abs(randomInt()) % words.length]);
   }
}
```

MockAnalyzer 和 MockTokenizer 用于词语分析相关的测试，相关的测试代码如下。

```
public class TestIndex extends RandomizedTest {
```

```
    public static void test1() throws IOException {
        Analyzer analyzer = new MockAnalyzer(LuceneTestCase.random(),
                MockTokenizer.SIMPLE, true);
        Directory rd = new RAMDirectory();
        IndexWriter w = new IndexWriter(rd, new IndexWriterConfig(analyzer));
    }

}
```

实现中文分词的 TokenizerFactory：

```
public class BigramTokenizerFactory extends TokenizerFactory{

    protected BigramTokenizerFactory(Map<String, String> args) {
        super(args);
    }

    @Override
    public Tokenizer create(AttributeFactory factory) {
        String dicPath = "./dic/";
        BigramDictioanry dict = BigramDictioanry.getInstance(dicPath);
        return new BigramTokenizer(factory,dict);
    }

}
```

使用 BaseTokenStreamTestCase 测试 BigramTokenizerFactory：

```
public class TestBigramTokenizerFactory extends BaseTokenStreamTestCase{
    public void testSimple() throws Exception {
        Reader reader = new StringReader("我购买了道具和服装。");
        TokenizerFactory factory =
            new BigramTokenizerFactory(new HashMap<String,String>());
        Tokenizer tokenizer = factory.create(newAttributeFactory());
        tokenizer.setReader(reader);
        assertTokenStreamContents(tokenizer,
            new String[] { "我", "购买", "了", "道具", "和", "服装", "。" });
    }
}
```

在 Es 中使用的 TokenizerFactory：

```
public class BigramESTokenizerFactory extends AbstractTokenizerFactory{

    public BigramESTokenizerFactory(IndexSettings indexSettings, String name,
            Settings settings) {
        super(indexSettings, name, settings);
    }
```

```
    @Override
    public Tokenizer create() {
        String dicPath = "./dic/";
        BigramDictioanry dict = BigramDictioanry.getInstance(dicPath);
        return new BigramTokenizer(dict);
    }

}
```

2.1.4　中文 AnalyzerProvider

编写支持中文的 AnalyzerProvider 类 CnAnalyzerProvider：

```
public class CnAnalyzerProvider
        extends AbstractIndexAnalyzerProvider<NgramAnalyzer>{
    private final NgramAnalyzer analyzer;

    @Inject
    public CnAnalyzerProvider(Index index, @IndexSettings Settings indexSettings,
            Environment env, @Assisted String name, @Assisted Settings settings){
        super(index, indexSettings, name, settings);
        //得到插件目录，词典文件放在插件目录的子目录下
        File pluginDir = env.pluginsFile();
        String dicPath=new File(pluginDir,"seg/dic").getPath();
        analyzer = new NgramAnalyzer(dicPath+"/");
    }

    @Override
    public NgramAnalyzer get() {
        return analyzer;
    }
}
```

注意，这里的注解“@Inject”是必需的。把生成出来的 seg.jar 放入插件路径下 D:\elasticsearch-5.1.2 \plugins\seg。

elasticsearch.yml 配置文件中增加中文分析器：

```
index:
  analysis:
    analyzer:
      cn:
          alias: [cn_analyzer]
          type: com.lietu.ds.CnAnalyzerProvider

index.analysis.analyzer.default.type : "com.lietu.ds.CnAnalyzerProvider"
```

可以使用 curl 测试分析器：

```
# curl -XGET 'localhost:9200/_analyze?pretty' -H 'Content-Type: application/json' -d'
{
   "analyzer" : "standard",
   "text" : "this is a test"
}'
```

也可以在 head 插件中测试这个分析器。索引下面有个 test analyze 选项。设置全局默认分词会看到效果，如果没有，还是会按照 Es 默认的一元分词。

可以先创建一个索引库，然后在这个索引上测试分词器。

查看分词效果的命令：

```
_analyze?text='我爱北京天安门'&analyzer=standard
_analyze?text='我爱北京天安门'&analyzer=cn
```

例如，news 索引使用如下的测试地址：

```
http://localhost:9200/news/_analyze?analyzer=cn&text='我爱北京天安门'
http://localhost:9200/news/_analyze?analyzer=standard&text='我爱北京天安门'
```


inquisitor（https://github.com/polyfractal/elasticsearch-inquisitor）是一个测试分词的插件。

index_analyzer 是代表这个字段建立索引时使用的分词方式，search_analyzer 代表对这个字段搜索时使用的分词。

mappings 配置的例子：

```
{
    "recruitinfo":{
        "properties":{
            "id":{
                "type":"string",
                "index":"not_analyzed"
            },
            "title":{
                "type":"string",
                "term_vector": "with_positions_offsets",
                "index_analyzer": "cn",
                "search_analyzer": "cn",
                "store":"yes"
            }
        }
    }
}
```

使用这个 mappings：

```
client.admin().indices().prepareCreate(indexName)
    .setSettings("... your JSON settings..")
        .addMapping(type, "... your mapping...")
```

2.1.5　字词混合索引

查询某些短语时，按字分词列和按词分词列的返回结果数量不一样，测试代码如下。

```
public static void searchField(String field,String keyWords ){
        MatchQueryBuilder qb = QueryBuilders.matchPhraseQuery(field, keyWords);

        Client client = getClient();
        SearchResponse searchResponse = client
                        .prepareSearch(ESConfig.indexName).setQuery(qb).execute()
                        .actionGet();
        SearchHits hits = searchResponse.getHits();

        long totalHits = hits.getTotalHits(); //得到结果总数
        System.out.println("totalHits:" + totalHits);
}
```

测试：

```
String keyWords = "一九九六年三月~二  OO 一年";
String field = "contentsS";
searchField(field ,keyWords);   //按字查询

field = "contents";
searchField(field ,keyWords);   //按词查询
```

输入相同的查询词返回结果数量不一样。

为了保证搜索的查全和查准，对全文查询列用单字索引和词索引两列索引同样的内容。对于标题，按字索引的列在 Mapping 中的定义如下。

```
"stitle":{
    "type":"string",
    "term_vector": "with_positions_offsets",
    "index_analyzer": "standard",
    "search_analyzer": "standard",
    "store":"yes"
},
```

按词索引的列在 Mapping 中的定义如下。

```
"title":{
    "type":"string",
    "term_vector": "with_positions_offsets",
    "index_analyzer": "cn",
    "search_analyzer": "cn",
    "store":"yes"
}
```

这里的 cn 是在配置文件中指定的，standard 是 Es 自带的。

打开 Index Metadata，可以看到索引库的结构。可以用 Java 代码修改索引库的结构，增加字索引列。

可以定义 Mapping，一列是按字索引，另一列是按词索引。D:\elasticsearch-1.0.0\config\mappings\news 目录下的 type1.json 内容如下。

```
{
    "type1":{
        "properties":{
            "title":{
                "type":"string",
                "term_vector": "with_positions_offsets",
                "index_analyzer": "cn",
                "search_analyzer": "cn",
                "store":"yes"
            },
            "body":{
                "type":"string",
                "term_vector": "with_positions_offsets",
                "index_analyzer": "cn",
                "search_analyzer": "cn",
                "store":"yes"
            },
            "stitle":{
                "type":"string",
                "term_vector": "with_positions_offsets",
                "index_analyzer": "standard",
                "search_analyzer": "standard",
                "store":"yes"
            },
            "sbody":{
                "type":"string",
                "term_vector": "with_positions_offsets",
                "index_analyzer": "cn",
                "search_analyzer": "cn",
                "store":"yes"
            }
        }
    }
}
```

使用 API 创建 json 内容：

```
XContentBuilder mapping = XContentFactory
    .jsonBuilder()
    .startObject()
```

```
        .startObject(indexType)
        //
        .startObject("properties")
        //
        .startObject("title")
        .field("type", "string")
        //start title
        .field("store", "yes")
        .field("analyzer", "cn")    //词
        //
        .endObject()
        //end title
        .startObject("postDate")
        //
        .field("type", "date").field("store", "yes")
        .field("index", "analyzed")
        //
        .endObject()
        //end post date
        .startObject("body")
        //
        .field("type", "string").field("store", "yes")
        //
        .field("index_analyzer", "standard")      //字
        .field("search_analyzer", "standard")
        .endObject() //end field
        .endObject() //end properties
        .endObject() //end index type
        .endObject();
```


为了保证连续匹配的文档得分高，把短语查询和普通的模糊查询组合成布尔查询，写法如下。

```
//短语查询
MatchQueryBuilder pqTitle = QueryBuilders.matchPhraseQuery(title, qString);
MatchQueryBuilder pqTitle2 = QueryBuilders.matchPhraseQuery(title2, qString);
MatchQueryBuilder pqBody = QueryBuilders.matchPhraseQuery(body, qString);
MatchQueryBuilder pqBody2 = QueryBuilders.matchPhraseQuery(body2, qString);

//普通模糊查询
QueryStringQueryBuilder fuzzyQb = new QueryStringQueryBuilder(qString)
            .field(title2).field(body2).field(title).field(body);

//把上面的查询组合到布尔查询得到最终的查询
QueryBuilder qb =
    queryBuilders.boolQuery().should(pqTitle2).should(pqTitle).
```

```
should(pqBody2).should(pqBody).should(fuzzyQb);
```

2.2 搜索英文

如果未指定所使用的分析器，那么默认使用的 Standard Analyzer 可以切分出英文单词。为了更深入地分析英文文章，可以专门开发针对英文的分析器。

2.2.1 句子切分

句子切分并不是一个简单的问题。标点符号“?”和“!”的含义比较单一。但是“.”有很多种不同的用法，并不一定是句子的结尾。例如，“Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group.”需要排除掉一部分情况。如果“.”是某个短语中间的一部分，则它不是句子的结尾。这里的“Mr. Vinken”是一个人名短语。如果这个人名正好不在词典中，则可以根据上下文识别规则识别出这个短语。


```
String text= "Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group.";
EnText enText = new EnText(text);
for(Sentence sent:enText){
    System.out.println(sent); //因为输入的是一个句子，所以这里只会打印出一个句子
}
```

Java 中的 BreakIterator 类已经包含了切分句子的功能。用它实现一个英文句子迭代器：

```
private final static class SentBreakIterator implements Iterator<Sentence> {
    String text;
    int start;
    int end;
    //根据英文标点符号切分
    static final BreakIterator boundary = BreakIterator
            .getSentenceInstance(Locale.ENGLISH);

    public SentBreakIterator(String t) {
        text = t;
        //设置要处理的文本
        boundary.setText(text);
        start = boundary.first(); //开始位置
        end = boundary.next();
    } //用于迭代的类

    @Override
    public boolean hasNext() {
        return (end != BreakIterator.DONE);
```

```
        }

        @Override
        public Sentence next() {
            String sent = text.substring(start, end);

            Sentence sentence = new Sentence(sent, start, end);
            start = end;
            end = boundary.next();
            return sentence;
        }
    }
```

BreakIterator 分得不太准确，所以我们自己写一个句子切分器。输入当前切分点，找下一个切分点的代码如下。

```
    public static int nextPoint(String text, int lastEOS) {
        int i = lastEOS;
        while (i < text.length()) {
            //跳过短语
            i = skipPhrase(text, i);

            //然后再找标点符号
            String toFind = eosDic.matchLong(text, i); //匹配标点符号词典
            if (toFind != null) {
                //判断是否有效的可切分点，如在括号中的标点符号不是有效的可切分点
                boolean isEndPoint = isSplitPoint(text, lastEOS, i);
                if (isEndPoint) {
                    return i + toFind.length();
                }
                i = i + toFind.length();
            } else { //没找到
                i++;
            }
        }
        return text.length(); //返回最大长度
    }
```

SentIterator 是一个用于迭代英文文本返回句子的内部类，实现代码如下。

```
    private final static class SentIterator implements Iterator<Sentence> {
        String text;
        int lastEOS = 0;

        public SentIterator(String t) {
            text = t;
```

```
        }

        @Override
        public boolean hasNext() {
            return (lastEOS < text.length());
        }

        @Override
        public Sentence next() {
            int nextEOS = EnSentenceSpliter.nextPoint(text, lastEOS);
            String sent = text.substring(lastEOS, nextEOS);
            Sentence sentence = new Sentence(sent, lastEOS, nextEOS);
            lastEOS = nextEOS;
            return sentence;
        }
    }
```


2.2.2 标注词性

一段英文：Cats never fail to fascinate human beings. They can be friendly and affectionate towards humans, but they lead mysterious lives of their own as well.

标注词性后的结果是：

Cats(n.) never fail(v.) to(prep) fascinate(v.) human(n.) beings(n.). They(pron.) can(aux.) be(v.) friendly(adj.) and(conj.) affectionate(adj.) towards(prep) humans(n.) but(conj.) they(n.) lead(v.) mysterious(adj.) lives(n.) of(prep.) their(n.) own(n.) as well(adv.).

这里用编码来表示词性。括号中的输出是词性编码。汉语中的量词在英语中没有的。例如，件、个、艘。英语中也有一些独有的词性，如冠词 a、an、the。英文词性编码表如表 2-2 所示。

表 2-2　英文词性编码表

代　码	名　称	代　码	名　称
n	名词	conj	连接词
adj	形容词	v	动词
adv	副词	num	数词
art	冠词	prep	介词
pos	所有格	punct	标点符号
pron	代词	int	感叹词
aux	情态助动词		

词性标注的流程图如图 2-3 所示。

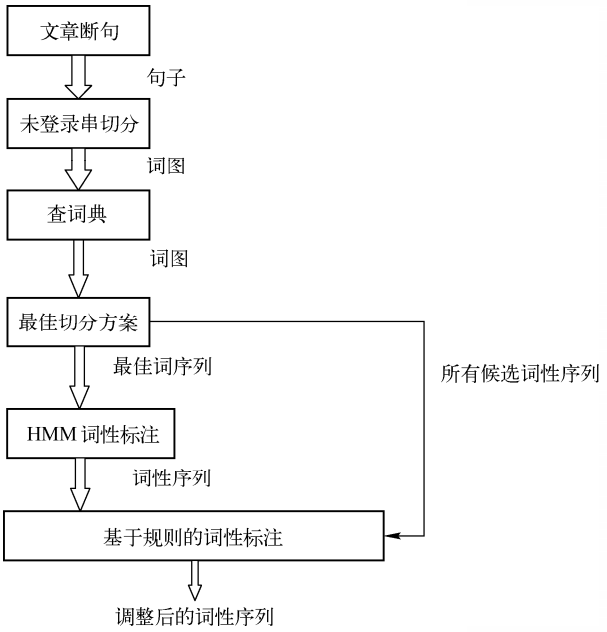

图 2-3　英文分词流程图

关于隐马尔可夫模型做词性标注在中文词性标注实现中已经介绍过了。英文词性标注语料库和中文词性标注语料库不一样，可以从 github 网站找到一些免费的资料。

标注规则如 I like it 对应的词性序列[r v r]。

```
key = new ArrayList<PartOfSpeech>();
key.add(PartOfSpeech.pron); //I
key.add(PartOfSpeech.v); //like
key.add(PartOfSpeech.pron); //it
posTrie.addProduct(key);
```

实现代码：

```
public static ArrayList<WordToken> getWords(Sentence sent){
    ArrayList<WordTokenInf> words = Segmenter.seg(sent); //先分词
    WordType[] tags = g.tag(words); //然后标注词性

    //再把词性和词本身结合起来，返回完整的词性标注结果
    int i=0;
    ArrayList<WordToken> tokens = new ArrayList<WordToken>();

    for(WordTokenInf w:words){
        WordToken t = new WordToken(w.baseForm,w.termText,w.start,w.end,tags[i]);
        ++i;
        tokens.add(t);
    }

    return tokens;
```

```
}
```

2.3 使用测试套件

随着应用程序变得更加复杂，正确的测试变得与以前一样重要。多年来，Elasticsearch提供了卓越的测试工具，以简化对依赖其搜索和分析功能的应用程序的测试。更具体地说，在项目中可能需要两种类型的测试。

- 单元测试：它们分别测试各个单元（如类），通常不需要运行 Elasticsearch 节点或集群。这类测试由 ESTestCase 和 ESTokenStreamTestCase 支持。
- 集成测试：测试完整的流程，通常至少需要一个正在运行的 Elasticsearch 节点（或集群，以强调更实际的场景）。这类测试由 ESIntegTestCase、ESSingleNodeTestCase 和 ESBackCompatTestCase 支持。

使用 Apache Maven 声明依赖。

```
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-test-framework</artifactId>
    <version>6.4.0</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.elasticsearch.test</groupId>
    <artifactId>framework</artifactId>
    <version>5.2.0</version>
    <scope>test</scope>
</dependency>
```


很可能需要在测试运行期间通过将 tests.security.manager 属性设置为 false 来关闭安全管理器。这可以通过直接将-Dtests.security.manager=false 参数传递给 JVM 或使用 Apache Maven 插件配置来完成。

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.19.1</version>
    <configuration>
        <argLine>-Dtests.security.manager=false</argLine>
    </configuration>
</plugin>
```

启用 3 个节点的 Elasticsearch 搜索集群。

```
@ClusterScope(numDataNodes = 3)
public class ElasticsearchClusterTest extends ESIntegTestCase {
```

```
}
```

尽管集群已经启动，但它没有任何索引，也没有预配置。下面添加一些测试背景来创建目录索引及其映射类型，使用 catalog-index.json 文件定义索引结构。catalog-index.json 文件内容如下。

```
{
  "settings": {
    "index" : {
      "number_of_shards" : 5,
      "number_of_replicas" : 2
    }
  },
  "mappings": {
    "books": {
      "_source" : {
        "enabled": true
      },
      "properties": {
        "title": { "type": "text" },
        "categories" : {
          "type": "nested",
          "properties" : {
            "name": { "type": "text" }
          }
        },
        "publisher": { "type": "keyword" },
        "description": { "type": "text" },
        "published_date": { "type": "date" },
        "isbn": { "type": "keyword" },
        "rating": { "type": "byte" }
       }
    },
    "authors": {
      "properties": {
        "first_name": { "type": "keyword" },
        "last_name": { "type": "keyword" }
      },
      "_parent": {
          "type": "books"
       }
    }
  }
}
```

测试代码如下。

```
@Before
public void setUpCatalog() throws IOException {
    try (final ByteArrayOutputStream out = new ByteArrayOutputStream()) {
        Streams.copy(getClass().getResourceAsStream("/catalog-index.json"),
            out);

        final CreateIndexResponse response = admin()
          .indices()
            .prepareCreate("catalog")
            .setSource(out.toByteArray())
            .get();

        assertAcked(response);
        ensureGreen("catalog");
    }
}
```


Elasticsearch 测试框架提供了 client()或 admin()方法，以及 getRestClient()方法，以便使用 Java REST 客户端实例。最好在每个测试运行之后清除集群，可以使用 cluster()方法来访问一些非常有用的操作，例如：

```
@After
public void tearDownCatalog() throws IOException, InterruptedException {
    cluster().wipeIndices("catalog");
}
```

总体而言，Elasticsearch 测试工具旨在实现两个目标：简化最常见的任务，并且很容易地进行验证、断言或期望。

首先，空索引应该没有文档，所以我们的第一个测试用例会明确地声明这个事实：

```
@Test
public void testEmptyCatalogHasNoBooks() {
    final SearchResponse response = client()
        .prepareSearch("catalog")
        .setTypes("books")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(QueryBuilders.matchAllQuery())
        .setFetchSource(false)
        .get();

    assertNoSearchHits(response);
}
```

这个很简单，接下来如何创建真实文档呢？Elasticsearch 测试框架具有广泛的、有用的方法来生成大多数任何类型的随机值。可以用它来创建一本书，将其添加到书目录索引中，并针对它写出查询。

```
@Test
public void testInsertAndSearchForBook() throws IOException {
    final XContentBuilder source = JsonXContent
        .contentBuilder()
     .startObject()
        .field("title", randomAsciiOfLength(100))
        .startArray("categories")
            .startObject().field("name", "analytics").endObject()
            .startObject().field("name", "search").endObject()
            .startObject().field("name", "database store").endObject()
        .endArray()
        .field("publisher", randomAsciiOfLength(20))
        .field("description", randomAsciiOfLength(200))
        .field("published_date", new LocalDate(2015, 02, 07).toDate())
        .field("isbn", "978-1449358549")
        .field("rating", randomInt(5))
        .endObject();

    index("catalog", "books", "978-1449358549", source);
    refresh("catalog");

    final QueryBuilder query = QueryBuilders
        .nestedQuery(
            "categories",
            QueryBuilders.matchQuery("categories.name", "analytics"),
            ScoreMode.Total
        );

    final SearchResponse response = client()
        .prepareSearch("catalog")
        .setTypes("books")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(query)
        .setFetchSource(false)
        .get();

    assertSearchHits(response, "978-1449358549");
}
```


除了类别以外的大部分图书属性是随机生成的，所以能够可靠地搜索它们。

Elasticsearch 测试支持开启了许多有趣的机会，不仅可以测试成功的结果，而且还可以模拟现实的集群行为和错误的条件（由 internalCluster()提供的帮助方法在此非常有用）。对于像 Elasticsearch 这样一个复杂的分布式系统，这样的测试的价值是无价的，所以利用可用的选项来确保部署到生产中的代码是稳健的，能够容错。例如，可以在运行搜索请求时关闭随机数据节点，并声明它们仍在处理中。

```
@Test
public void testClusterNodeIsDown() throws IOException {
    internalCluster().stopRandomDataNode(); //关闭随机数据节点

    final SearchResponse response = client()
        .prepareSearch("catalog")
        .setTypes("books")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(QueryBuilders.matchAllQuery())
        .setFetchSource(false)
        .get();

    assertNoSearchHits(response);
}
```

2.4 本章小结

为了让 Elasticsearch 能够搜索中文，除了字词混合索引的方法，还可以使用 IKAnalyzer。

关于英文文章分析，还可以使用 OpenNLP 解析句子。

日文分词可以使用 Kuromoji(https://github.com/atilika/kuromoji)。

韩文分词可以使用 Korean Analysis for ElasticSearch(https://github.com/usemodj/elasticsearch-analysis-korean)。

管理搜索集群

默认 Elasticsearch 是使用 Netty 作为 HTTP 的容器的，由于 Netty 并没有权限模块，所以默认 Es 没有任何的权限控制，直接通过 HTTP 就可以进行任何操作，除非把 HTTP 禁用。但如果使用 elasticsearch-jetty 插件，就可以使用 jetty 自带的权限管理进行一些权限的控制，同时也可以支持通过 HTTPS 协议来访问 Es，还有就是支持 gzip 压缩响应信息。

3.1 节点类型

启动一个 Elasticsearch 实例，就是启动一个节点。连接在一起的节点的集合称为一个集群。如果正在运行 Elasticsearch 的单个节点，那么就是一个节点组成的集群。

默认情况下，集群中的每个节点都可以处理 HTTP 和 Transport 流量。Transport 层专用于节点和 Java TransportClient 之间的通信，而 HTTP 层仅由外部 REST 客户端使用。

所有节点都知道集群中的所有其他节点，并能够将客户端请求转发到适当的节点。

除此之外，每个节点都有一个或多个目的。

- 符合 Master 资格的节点：一个节点的 node.master 设置为 true（默认值），这使得它有资格被选为控制集群的主节点。
- 数据节点：node.data 设置为 true（默认值）的节点。数据节点保存数据并执行数据相关操作，如 CRUD、搜索和聚合。
- 摄取节点：node.ingest 设置为 true（默认值）的节点。摄取节点能够将摄取流水线应用于文档，以便在索引之前转换和丰富文档。因为摄取负载沉重，所以建议使用专门的摄取节点并将主节点和数据节点标记为 node.ingest:false 是有意义的。

默认情况下，Elasticsearch 集群中的每个节点都有成为主节点的资格，也都存储数据，还可以提供查询服务。这对于小型集群来说非常方便，但随着集群的发展，考虑将专用主节点与专用数据节点分开是很重要的。

3.2 管理集群

reroute 命令允许显式地执行包含特定命令的集群重路由分配命令。例如，分片可以从一个节点移动到另一个节点，可以取消分配，或者可以在特定节点上显式地分配未分配的分片。

以下是一个简单的 reroute API 调用的简短示例。

```
POST /_cluster/reroute
{
    "commands" : [
        {
            "move" : {
                "index" : "test", "shard" : 0,
                "from_node" : "node1", "to_node" : "node2"
            }
        },
        {
          "allocate_replica" : {
                "index" : "test", "shard" : 1,
                "node" : "node3"
          }
        }
    ]
}
```


当分片丢失时，也可以尝试使用 reroute 命令找回丢失的分片。

从 Es 5.5 开始，reroute 命令已分成为两个不同的命令 allocate_replica 和 allocate_empty_primary。

引入了一个新的 allocate_stale_primary 命令。新的 allocate_replica 命令对应于 allow_primary 设置为 false 的旧分配命令。新的 allocate_empty_primary 命令对应于 allow_primary 设置为 true 的旧分配命令。

3.3 写入权限控制

如果能让 Es 提供只读，而不可以写入的端口号，就能够实现权限控制。一个方法是把 Es 部署在 Jetty（https://github.com/sonian/elasticsearch-jetty）中。

另外一个方法是把 Nginx 放在 Es 前端。只允许 GET 请求的一个例子配置如下。

```
worker_processes    1;
pid                 nginx.pid;

events {
    worker_connections    1024;
}

http {

  server {

    listen          8080;
```

```
        server_name    search.example.com;

        error_log      elasticsearch-errors.log;
        access_log     elasticsearch.log;

        location / {
          if ($request_method !~ "GET") {
            return 403;
            break;
          }

          proxy_pass http://localhost:9200;
          proxy_redirect off;

          proxy_set_header    X-Real-IP    $remote_addr;
          proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
          proxy_set_header    Host $http_host;
        }

      }

    }
```


使用这个配置：

```
$ nginx -c path/to/this/file
```

然后测试它：

```
curl -i -X GET http://localhost:8080/_search -d '{"query":{"match_all":{}}}'
HTTP/1.1 200 OK

curl -i -X POST http://localhost:8080/test/test/1 -d '{"foo":"bar"}'
HTTP/1.1 403 Forbidden

curl -i -X DELETE http://localhost:8080/test/
HTTP/1.1 403 Forbidden
```

安装 X-Pack 插件后，所有对 Es 的访问都增加了安全机制，即需要用户名和密码，默认分别为：elastic 和 changeme。使用 sense 插件访问时可以输入，如果是使用 curl 等方式访问，则需要在 HTTP 的 header 中增加 Authentication 参数。

安全的客户端可以参考 https://github.com/elastic/found-shield-example。

3.4 使用 X-Pack

X-Pack 插件包含安全、警报、监视、报告和图形功能。安装 X-Pack 插件：

```
bin/elasticsearch-plugin install x-pack
```

列出所有加载的插件：

```
bin/elasticsearch-plugin list
```

可以看到 X-pack 已经安装。
如果不需要该插件，也可以删除：

```
bin/elasticsearch-plugin remove x-pack
```

在使用 crul 命令时加入认证参数：-user username:password。
使用默认的用户名 elastic，默认密码 changme 访问索引：

```
curl -user elastic:changeme -XPUT ‘localhost:9200/idx’
```

3.5 快照

为了保证数据的完整性，可以使用快照功能把历史数据存入 Hadoop 集群的分布式文件系统中。HDFS 存储库插件增加了使用 HDFS 文件系统作为 Snapshot/Restore 存储库的支持。
该插件可以使用插件管理器安装：

```
sudo bin/elasticsearch-plugin install repository-hdfs
```

该插件必须安装在群集中的每个节点上，每个节点必须在安装后重新启动。
该插件也可以下载安装：

```
https://artifacts.elastic.co/downloads/elasticsearch-plugins/repository-hdfs/repository-hdfs-5.3.0.zip
```

HDFS 快照/恢复插件是针对最新的 Apache Hadoop 2.x（目前为 2.7.1）构建的。如果正在使用的发行版与 Apache Hadoop 不兼容，请考虑使用自己的插件文件替换插件文件夹中的 Hadoop 库。
安装后，通过 REST API 定义 HDFS 仓库的配置：

```
PUT /_snapshot/my_backup
{
"type": "hdfs",
"settings": {
        "path": "/path/on/hadoop",
        "uri": "hdfs://hadoop_cluster_domain:[port]",
        "conf_location":"/hadoop/hdfs-site.xml,/hadoop/core-site.xml",
        "user":"hadoop"
    }
}
```

为索引创建快照：

```
PUT /_snapshot/my_backup/snapshot_1?wait_for_completion=true
```

可以使用以下命令恢复快照：

```
POST /_snapshot/my_backup/snapshot_1/_restore
```

Lucene 索引可以增量备份，以便减少硬盘空间占用。

3.6 Zen 发现机制

Elasticsearch 集群默认使用 Zen Discovery（Zen 发现机制）管理。

Zen 发现机制是 Elasticsearch 默认的内建模块。它提供了多播和单播两种发现方式，能够很容易地扩展至云环境。

Zen 发现机制是和其他模块集成的，如所有节点间通信必须用 Transport 模块来完成。Transport 这层是自己可以扩展的，thrift 也是一个插件。

Elasticsearch 运行时会启动两个探测进程。一个进程用于从主节点向集群中其他节点发送 ping 请求来检测节点是否正常可用。另一个进程的工作反过来，由其他节点向主节点发送 ping 请求来验证主节点是否正常且忠于职守。

一个集群有唯一的名字，包含一个或多个节点。集群会在所有的节点中自动选择一个作为主节点，如果主节点宕机了，则会自动选择另一个节点作为主节点。一个经典的主节点选举算法是同行评审出版算法（Peer-reviewed Published Algorithm）。

Elasticsearch 采用了一个简单的方法选出主节点：它根据编号来选择节点，较小的编号更有可能成为主节点。DiscoveryNode 类中记录了节点编号。选举算法的实现代码在 ElectMasterService.electMaster()方法中。

为了避免一个集群中存在不同的主节点，需要合理地设置 elasticsearch.yml 配置文件。

假设可以成为集群一部分的 Es 节点的数量（Es 进程而不是物理机器的数量）是 N，那么在一个有 $N>2$ 个节点的集群上，可以设置 discovery.zen.minimum_master_nodes 的值不小于$(N/2)+1$。

理想的拓扑结构有 3 个专用的主节点（即 master: true 并且 data:false），并且 discovery.zen。minimum_master_nodes 设置为 2。这样无论有多少个数据节点应该是集群的一部分，都不需要改变设置。

每个文档都保存在单独的主分片里。当对一个文档做索引时，首先对主分片做索引，然后在所有主分片的副本里做索引。默认一个索引有 5 个主分片，可以调整主分片的数量以控制一个索引中容纳文档的数量。索引创建之后，不可以更改主分片数。即使只在一台机器上安装 Es，也可能会有 5 个独立的索引库。

每个主分片可以有 0 个或多个副本。副本是主分片的复制品，有以下两个作用。

（1）提高容错能力：如果主分片宕机，副本分片可以被提升至主分片。

（2）提高性能：搜索访问可以分布在主分片和副本分片之间。

默认每个主分片有一个副本分片，但副本分片数量可以在已经存在的索引上动态调整。在同一个节点上，副本分片不会被当成主分片启动。

举例说明索引分片的用处：第一台机器中存放索引分片 a、b、c，第二台机器中存放索引分片 a、b、d，第三台机器中存放索引分片 b、c、d。提升索引整体容量的同时，提升性能和容错能力。

新增一个节点，Elasticsearch 会自动把索引数据同步到这个新增的节点上。控制界面中显示的紫色的块表示正在迁移这部分数据。

主控节点管理 shard 的分配。当新机器进来或有旧机器失效时，就会重新分配 shard。

依赖注入（Dependency Injection，DI）很好，因为一个节点有很多个索引，每个索引有很多个分片，每个分片是不同的 Guice 模块。Elasticsearch 使用 Google 开源的依赖注入框架 Guice（https://github.com/google/guice），没有使用 Spring 实现依赖注入的原因是：Spring 需要配置文件，用起来太笨重。Elasticsearch 直接把 guice 的源码放到自己的 org.elasticsearch.common.inject 包内。

3.7 联合搜索

使用跨群集搜索来执行联合搜索。自从 Elasticsearch 5.3.0 以来，可以通过 search.remote 命名空间下的集群更新设置 API 注册远程集群。每个集群都由集群别名和用于发现属于远程集群的其他节点的种子节点列表进行标识，如下所示。

```
PUT _cluster/settings
  {
   "persistent": {
      "search": {
         "remote": {
            "cluster_one": {
               "seeds": ["remote_node_one:9300"]
            },
            "cluster_two": {
               "seeds": ["remote_node_two:9300"]
            }
         }
      }
   }
  }
```


一旦注册了一个或多个远程集群，就可以使用_search API 对其索引执行搜索请求了。与本地索引相反，远程索引必须增加群组别名前缀来消除歧义，如 cluster_two:index_test。

每当搜索请求扩展到远程集群上的索引时，协调节点通过每个集群发送一个_search_shards 请求来解析远程集群上这些索引的分片。一旦获取了分片和远程数据节点，就像上面所述的本地集群上的任何其他搜索一样，执行搜索，使用完全相同的代码路径，显著提高了可测试性和鲁棒性。

可以通过 search.remote.connect 设置来控制哪些节点能够作为跨集群搜索请求的协调节点。这对于控制集群中的哪些节点可以向远程集群发送请求是有用的。如果不允许连接到远程集群的节点接收到涉及远程集群的搜索请求，则会返回错误。

3.8 缓存

响应查询需要花费 CPU 时间、内存、时间。增加集群的处理能力有助于解决这个问

题，但过度配置可能会非常昂贵。缓存经常是从优化工具箱中拉出的第一个工具。

缓存使用的内存总是有限的。需要使用一种算法来检测和替换不值得的东西。有一些算法用于缓存项替换，包括：

- Least Recently Used（LRU）：最近最少使用。
- Least Frequently Used（LFU）：最不经常使用。
- First In First Out（FIFO）：先进先出。

最受欢迎的一个算法是最近最少使用（LRU）算法。研究表明，相比旧项目来说，更可能使用新的项目。LRU 就是基于这个观察的。该算法保持跟踪项目的最后访问时间。它清除有最古老的访问时间戳的项目。

虽然较旧版本的 Elastisearch 缓存了所有可缓存的项目，但较新版本在默认情况下相当有选择性。那么 Elasticsearch 支持哪些缓存，利用它的最佳方法是什么呢？

Elasticsearch 支持三种类型的缓存：节点查询缓存、分片请求缓存和字段数据缓存。

- 节点查询缓存是节点上所有分片共享的 LRU 缓存。它缓存在过滤器上下文中使用的查询的结果，在以前的 Elasticsearch 版本中，由于这个原因，它被称为过滤器缓存。过滤器上下文中的子句用于包含（或排除）结果集中的文档，但不影响评分。此外，Elasticsearch 观察到，许多过滤器计算速度非常快，特别是对于小的段，而其他过滤器则很少见。为了减少流失，节点缓存只包括以下过滤器：在最近的 256 个查询中被多次使用属于有超过 10 000 个文档的段（或占文档总数 3%的过滤器，以较大者为准）。
- 分片请求缓存为每个分片独立地缓存查询结果，也使用 LRU 替换。默认情况下，请求缓存也限制子句：只缓存大小为 0 的请求（如聚合、计数和建议）。如果认为应该缓存查询，就可以在请求中添加“request_cache = true”标志。不是所有的子句都将被缓存。包含“now”的 DateTime 子句不会被缓存，在每次更新分片时都让分片请求缓存失效，这可能导致在频繁更新的索引中性能不佳。
- 字段数据缓存是指当 Elasticsearch 计算字段上的聚合时，它会将所有字段值加载到内存中。因此，Elastisearch 中的计算聚合可以是查询中最昂贵的操作之一。字段数据缓存在计算聚合时保存字段值。虽然 Elastisearch 不跟踪命中/未命中率，但建议将其设置为足够大，以便将所有值保存在内存中。

有许多集成可用于监控 Elasticsearch 缓存。Sematext 和 Datadog 是一些比较常见的，但如果只需要在开发过程中进行检查呢？

Elasticsearch 提供了许多方法来检查缓存利用率，但_cat 节点 API 会在一次调用中给出上述所有的值：

```
GET_cat/nodes?v&h=id,queryCacheMemory,queryCacheEvictions,requestCacheMemory,requestCacheHitCount,
requestCacheMissCount,flushTotal,flushTotalTime
```

3.9 本章小结

除了 X-Pack，还可以使用 Kibana 管理搜索集群。

源 码 分 析

首先介绍 Lucene 源码，然后介绍 Elasticsearch 源代码。

4.1 Lucene 源码分析

可以使用如下命令从 github.com 得到 Lucene 最新的源码。

```
#git clone https://github.com/apache/lucene-solr.git
```

4.1.1 Ivy 管理依赖项

Lucene 源代码采用 Ant 构建，使用了 Apache Ivy（http://ant.apache.org/ivy/）管理 jar 文件之间的依赖关系。

Ivy 特有的文件是 ivy.xml 和一个 Ivy 设置文件。ivy.xml 文件中列举了项目的所有依赖项。例如，nutch 依赖 HttpClient：

```
<dependency org="commons-httpclient" name="commons-httpclient"
        rev="3.1" conf="*->master" />
```

Ivy 依赖于 Ant，所以需要先安装 Ant，然后下载 Ivy，将它的 jar 文件复制到 Ant 的 lib 下，就可以在 Ant 里使用 Ivy 进行依赖管理了。

将 Apache Lucene-Solr 导入 Eclipse：

```
ant compile
ant eclipse
```

然后就能通过选项导入 Eclipse 了：File -> Import -> Existing Projects Into workspace。

4.1.2 源码结构介绍

Lucene 源码分为核心包和外围功能包。核心包实现搜索功能，外围功能包实现高亮显示等辅助功能。Lucene 源码的核心包中包括 7 个基本的功能子包，每个包完成特定的功能。

最基本的是索引管理包（org.apache.lucene.index）和检索管理包（org.apache.lucene.search）。索引管理包实现索引建立、删除等。检索管理包根据查询条件，检索得到结果。

索引管理包调用数据存储管理包（org.apache.lucene.store），主要包括一些底层的 I/O 操作。同时也会调用一些公用的算法类（org.apache.lucene.util）。编码管理包（org.apache.lucene.codecs）用于方便自定义索引的编码和结构。文档结构包（org.apache.lucene.document）用于描述索引存储时的文档结构管理，类似于关系型数据库的表结构。

查询分析器包（org.apache.lucene.queryParser）实现查询语法，支持关键词间的运算，如与、或、非等。语言分析器（org.apache.lucene.analysis）主要用于对放入索引的文档和查询词切词，支持中文主要是扩展此类。

4.2 Gradle

Elasticsearch 源代码使用 Gradle 构建。

可以下载二进制文件来安装 Gradle：

```
https://services.gradle.org/distributions/gradle-3.5-bin.zip
```


打开控制台并运行 gradle -v 命令显示版本来验证安装，例如：

```
$ gradle –v
```

可以在 Gradle 构建中使用标准和定制的 Ant 任务，就像在 Ant 本身中使用的那样。另外，可以导入现有的 Ant 脚本。就像这样简单：

```
ant.importBuild 'build.xml'
```

为了构建 Elasticsearch5 的源代码，需要安装版本 2.13 的 Gradle。为了创建发布包，只需在复制的目录中运行 gradle assemble 命令即可。

```
#git clone https://github.com/elastic/elasticsearch.git
```

每个项目的发布包创建在该项目的 build/distributions 目录下。

从源代码构建 Elasticsearch：

```
gradle build
```

运行一个测试用例：

```
gradle test -Dtests.class=org.elasticsearch.package.ClassName
gradle test "-Dtests.class=*.ClassName"
```

运行包和子包中所有的测试用例：

```
gradle test "-Dtests.class=org.elasticsearch.package.*"
```

4.3 Guice

Elasticsearch 使用 Guice 的 Provider 类创建和返回 Analyzer 对象。

使用 Provider 的例子：

```
public interface MyInterface {

    String foobar();

}

public class MyClass implements MyInterface {

    private String providerName;

    public MyClass(String providerName) {
        this.providerName = providerName;
    }

    @Override
    public String foobar() {
        return String.format("Hi! I am [%s], " + "and I was instantiated using [%s]", getClass().getSimpleName(), providerName);
    }

}
```


MyInterfaceProvider 提供 MyClass 的实例：

```
import com.google.inject.Provider;

public class MyInterfaceProvider implements Provider<MyClass> {

    @Override
    public MyClass get() {
        return new MyClass(getClass().getSimpleName());
    }

}
```

Module 的子类建立绑定：

```
import com.google.inject.AbstractModule;

public class MyModule extends AbstractModule {

    @Override
    protected void configure() {
        //绑定 MyInterface 接口到 Provider 子类
        bind(MyInterface.class).toProvider(MyInterfaceProvider.class);
    }
```

```
}
```

使用 Guice 得到 MyInterface 的对象：

```
import com.google.inject.Guice;
import com.google.inject.Injector;

public class ProviderSample {

    public static void main(String[] args) {
        Injector injector = Guice.createInjector(new MyModule());
        MyInterface myObject = injector.getInstance(MyInterface.class);

        System.out.println(myObject.foobar());
    }
}
```

运行 ProviderSample 输出：

```
Hi! I am [MyClass], and I was instantiated using [MyInterfaceProvider]
```


4.4 Joda-Time

Elasticsearch 内部使用 Joda-Time（http://www.joda.org/joda-time/）处理日期和时间。Joda-Time 提供了 date 和 time 类的替换。使用 Joda-Time 解析日期字符串的代码如下。

```
DateTimeFormatterBuilder builder = new DateTimeFormatterBuilder();
DateTimeParser[] parsers = new DateTimeParser[3];
parsers[0] = DateTimeFormat.forPattern("MM/dd/yyyy")
        .withZone(DateTimeZone.UTC).getParser();
parsers[1] = DateTimeFormat.forPattern("MM-dd-yyyy")
        .withZone(DateTimeZone.UTC).getParser();
parsers[2] = DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss")
        .withZone(DateTimeZone.UTC).getParser();
builder.append(
        DateTimeFormat.forPattern("MM/dd/yyyy")
                .withZone(DateTimeZone.UTC).getPrinter(), parsers);

DateTimeFormatter formatter = builder.toFormatter();
long millis = formatter.parseMillis("2009-11-15 14:12:12");
System.out.println(millis);
```

Elasticsearch 中的 StrictISODateTimeFormat 这个类从 Joda 几乎相同地复制过来了，Joda-Time 中该类被命名为 ISODatetimeFormat。然而，在几个方法中修改很大，日期年份至少为 *n* 位数，所以像“5”这样的年份是无效的，必须是“0005”。

创建时间对象：

```
MutableDateTime dateTime = new MutableDateTime(3000, 12, 31, 23, 59, 59, 999,
                                DateTimeZone.UTC);
System.out.println(dateTime);
```

4.5 Transport

默认节点的连接数量是 13 个：

```
ConnectionProfile profile =
    TcpTransport.buildDefaultConnectionProfile(Settings.EMPTY);
assertEquals(13, profile.getNumConnections());

//节点之间的 ping 连接个数 1 个
assertEquals(1, profile.getNumConnectionsPerType(TransportRequestOptions.Type.PING));
//典型的搜索和单 doc 索引，默认个数 6 个
assertEquals(6, profile.getNumConnectionsPerType(TransportRequestOptions.Type.REG));
//集群状态的发送，默认个数 1 个
assertEquals(1, profile.getNumConnectionsPerType(TransportRequestOptions.Type.STATE));
//做数据恢复 recovery，默认个数 2 个
assertEquals(2, profile.getNumConnectionsPerType(TransportRequestOptions.Type.RECOVERY));
//用于批量请求，默认个数 3 个
assertEquals(3, profile.getNumConnectionsPerType(TransportRequestOptions.Type.BULK));
```


4.6 线程池

每个 Elasticsearch 节点内部都维护着多个线程池，如 index、search、warmer、bulk 等，用户可以修改线程池的类型和大小。

通过如下命令查看线程池情况：

```
http://localhost:9200/_nodes/stats?pretty
```

4.7 模块

Elasticsearch 内置的模块如下。

- transport-netty4：网络通信。
- aggs-matrix-stats：矩阵聚合在多个字段上工作，并产生一个矩阵作为输出。
- analysis-common：包含一些常用的 TokenFilter。
- ingest-common：实现摄取的公用类。
- lang-expression：实现表达式脚本引擎。
- lang-mustache：mustache 脚本引擎。

- lang-painless：painless 脚本引擎。
- parent-join：父连接。
- percolator：实现注册查询功能。
- reindex：重新索引 elasticsearch 数据。
- repository-url：BlobStoreRepository 的只读基于 URL 的实现。

4.8 Netty

Netty（http://netty.io/）是一个 NIO 客户端服务器框架。Elasticsearch 采用 Netty 实现 HTTP 异步通信协议。

Elasticsearch 中的服务器端 Netty4HttpServerTransport 位于 transport-netty4 模块。客户端的 PreBuiltTransportClient 也依赖于 transport-netty4 模块。

Netty 是 Reactor 设计模式的一个实现。Reactor 设计模式是用于处理由一个或多个输入同时发送到服务处理程序的服务请求的事件处理模式。然后，服务处理器多路复用输入的请求并将其同步分派到相关联的请求处理程序。

ServerBootStrap 负责引导服务器启动 NIO 服务，使用 ServerBootStrap 的代码如下。


```
int workerCount=1;

ServerBootstrap serverBootstrap = new ServerBootstrap();
ThreadFactory f = new DefaultThreadFactory("thread pool"); //守护线程工厂
//Reactor 单线程模型
//I/O 事件作为一个触发器，网络请求事件在 NioEventLoop 中进行处理
serverBootstrap.group(new NioEventLoopGroup(workerCount, f));
```

java.nio.channels.SelectorProvider 在 Linux 下实现了基于 epoll 的事件通知工具。epoll 工具在 Linux 2.6 和更高版本的内核中可用。Netty 封装了对 SelectorProvider 的使用。

4.9 分布式

Elasticsearch 使用主节点管理集群。主节点是集群中唯一可以更改集群状态的节点。这意味着如果主节点重新启动或关闭，那么将无法对群集进行任何更改。任何一个时刻集群中只能有一个主节点，但为了避免单点失败，需要有多个候选主节点。

Elasticsearch 采用了一个简单的方法选出主节点：它根据编号来选择节点，较小的编号更有可能成为主节点。DiscoveryNode 类中记录了节点编号。选举算法的实现代码在 ElectMasterService.electMaster()方法中。

为了避免一个集群中存在不同的主节点，需要合理设置 elasticsearch.yml 配置文件。

假设可以成为集群一部分的 Es 节点的数量（Es 进程而不是物理机器的数量）是 N，那么在一个有 $N > 2$ 个节点的集群上，可以设置 discovery.zen.minimum_master_nodes 的值不小于$(N/2)+1$。

理想的拓扑结构是有 3 个专用的主节点（即 master: true 并且 data:false），并且 discovery.zen。minimum_master_nodes 设置为 2。这样无论有多少个数据节点应该是集群的

一部分，都不需要改变设置。

4.10 本章小结

ant 可以自动化打包逻辑。maven 也可以自动化打包，相比于 ant，它多做的事情是帮你下载 jar 包。但 maven 的打包逻辑太死板，定制起来太麻烦，不如 ant 好用。gradle 又能自动下 jar 包，又能自己写脚本。

Elasticsearch 早期的版本使用 JGroups 实现多播。JGroups 就是一个用于方便集群开发的组件。它依赖组播。

Netty 是一个高性能、事件驱动的、异步的非堵塞的 I/O 框架。默认 Elasticsearch 使用 Netty 作为 HTTP 的容器，由于 Netty 并没有权限模块，所以默认 Elasticsearch 没有任何的权限控制，直接通过 HTTP 就可以进行任何操作，除非把 HTTP 禁用。但如果使用 elasticsearch-jetty 插件，就可以使用 jetty 自带的权限管理进行一些权限的控制，同时也可以支持通过 HTTPS 协议来访问 Es，还有就是支持 gzip 压缩响应信息。

Elasticsearch 直到版本 5 为止，仍然采用 HTTP/1.1 协议，还没有采用性能更好的 HTTP/2 协议。

关于 Lucene 的 Java Doc 说明文档见 http://lucene.apache.org/core/documentation.html。

Cutting 在 1999 年写 Lucene 以前，用 C++开发过搜索引擎。Lucene 是他写的第一个 Java 软件。在后来的 10 多年里，Lucene 越来越流行，成为开源组织 Apache 基金会的项目，并在维基百科网站等项目中得到广泛使用。Cutting 后来开发 MapReduce 的 Java 版本 Hadoop 也同样成功。Cutting 因此进入 Apache 基金董事会，并在 2010 年成为董事会主席。

可以使用 Luke（https://github.com/DmitryKey/luke）分析本地硬盘上的 Elasticsearch 索引，使用 luke.bat 或 luke.sh 运行 luke，然后就可以在/indexname/0/index/这样的路径中打开索引了。

搜索相关性

评估查询词和文档的相关性当前有 BM25 检索模型和学习评分两种流行的方法。

5.1 BM25 检索模型

可以认为打上了和查询词同样标签的文档是相关文档，但在很多时候，猜测文档是否有相关内容是没有把握的，所以用概率来量化这种不确定性。把信息检索作为分类问题，一类是相关文档 R，还有一类是无关的文档 NR。根据贝页斯判别规则，如果 $P(R|D)>P(\mathrm{NR}|D)$，则 D 是相关的文档。如果 $P(R|D)<P(\mathrm{NR}|D)$，则 D 是不相关的文档。例如，$P(R|D)=0.8$，$P(\mathrm{NR}|D)=0.2$，则 D 是和用户查询相关的文档。如图 5-1 所示，把“新生婴儿入户须知”这个索引库中的文档分成相关文档或不相关文档。

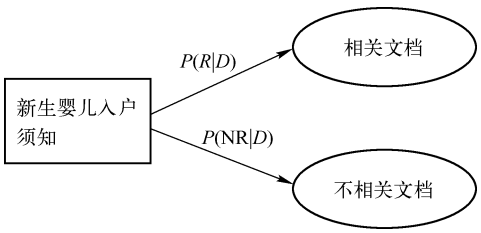

图 5-1　把信息检索看成分类问题

知道相关文档集合就能够计算出 $P(D|R)$。例如，如果知道某个词在相关文档集合中频繁出现，然后，给定一个新文档，就能直接计算出这个文档中词的组合有多大可能在相关文档集合中。

使用贝页斯公式估计概率：

$$P(R\mid D)=\frac{P(D\mid R)P(R)}{P(D)}$$

比较 $P(R|D)$和 $P(\mathrm{NR}|D)$的值。如果满足 $P(D|R)\ P(R)>P(D|\mathrm{NR})\ P(\mathrm{NR})$，就把文档分到相关类。把一个文档分类成相关的条件可以写成：

$$\frac{P(D\mid R)}{P(D\mid \mathrm{NR})}>\frac{P(\mathrm{NR})}{P(R)}$$

左边的公式称似然比，需要计算 $P(D|R)$和 $P(D|\text{NR})$。为了简化计算，把文档表示成词的组合，用词概率估计 $P(D|R)$和 $P(D|\text{NR})$。

用一个二值特征的向量表示文档的特征，表示文档中出现或不出现某个词。把文档表示成二项特征组成的向量。$D=(d_1, d_2, \cdots, d_t)$，如果词 i 出现在文档中，则 $d_i=1$，否则为 0。如果假设词都是独立出现的，则 $P(D|R)$可以用词概率的乘积 $\prod_{i=1}^{t} P(d_i \mid R)$ 计算。因为这个模型假设词独立出现，而且使用文档的二项特征，所以称二项独立模型。

假设索引库包含 5 个词，某文档 D 根据二元假设，表示为{1,0,1,0}，其含义是这个文档出现了第 1 个、第 3 个和第 5 个词，但不包含第 2 个和第 4 个词。

用 P_i 来代表第 i 个词在相关文档集合内出现的概率，于是在已知相关文档集合的情况下，文档 D 相关的概率为：

$$P(R|D)=P_1*(1-P_2)*P_3*(1-P_4)*P_5$$

其中的 $1-P_2$ 代表了第 2 个词不出现在相关文档的概率，因为 $P(t_2|R) + P(\neg\ t_2|R)=1$。

为了计算 $P(D|\text{NR})$，假设用 S_i 代表第 i 个词语或单词在不相关文档集合内出现的概率，于是在已知不相关文档集合的情况下，观察到文档 D 的概率为：

$$P(D|\text{NR})=S_1*(1-S_2)*S_3*(1-S_4)*S_5$$

例如，查询为：“信息 检索 教程”，所有词项的在相关、不相关情况下的概率 p_i、s_i 如表 5-1 所示。

表 5-1 p_i和 s_i的值

词　项	信　息	检　索	教　材	教　程	课　件
R 中的概率 p_i	0.8	0.9	0.3	0.32	0.15
NR 中的概率 s_i	0.3	0.1	0.35	0.33	0.10

假设文档 D_1 中只有 2 个词：“检索 课件”，则

$$P(D|R)=(1-0.8)*0.9*(1-0.3)*(1-0.32)*0.15$$

$$P(D|\text{NR})= (1-0.3)*0.1*(1-0.35)*(1-0.33)*0.10$$

$$P(D|R)/P(D|\text{NR})=4.216$$

返回到似然比。使用 P_i 和 S_i 得到分值：

$$\frac{P(D \mid R)}{P(D \mid NR)} = \prod_{i:d_i=1} \frac{p_i}{s_i} \prod_{i:d_i=0} \frac{1-p_i}{1-s_i}$$

其中 $\prod_{i:d_i=1}$ 的意思是在文档向量中值为 1 的词对应的乘积。转换把上面的公式：

$$\prod_{i:d_i=1} \frac{p_i}{s_i} \prod_{i:d_i=0} \frac{1-p_i}{1-s_i} = \prod_{i:d_i=1} \frac{p_i}{s_i} \left(\prod_{i:d_i=1} \frac{1-s_i}{1-p_i} \prod_{i:d_i=1} \frac{1-p_i}{1-s_i} \right) \prod_{i:d_i=0} \frac{1-p_i}{1-s_i}$$

$$= \prod_{i:d_i=1} \frac{p_i(1-s_i)}{s_i(1-p_i)} \prod_{i} \frac{1-p_i}{1-s_i}$$

第二项在所有定义向量维度的词上运算，因此对任何文档来说，值都是一样的。对文档评分时可以忽略此项。

因为多个很小的数乘起来可能导致精度丢失或者向下溢出成为 0，所以对计算公式取

log。这样评分公式变成了：

$$\sum_{i:d_i=1} \log \frac{p_i(1-s_i)}{s_i(1-p_i)}$$

如果存在相关性反馈可以得到相关文档和无关文档集合。也就是说，给定用户查询，如果可以确定哪些文档构成了相关文档集合，哪些文档构成了不相关文档集合，可以利用表 5-2 所列出的数据来估算单词概率。

表 5-2　某个查询的词出现情况的相关表

	相 关 文 档	不相关文档	文 档 总 数
d_i=1	r_i	n_i-r_i	n_i
d_i=0	$R-r_i$	$N-n_i-R+r_i$	$N-n_i$
文档总数	R	$N-R$	N

表中第 3 行的 N 为文档集合总共包含的文档个数，R 为相关文档的个数，于是 N-R 就是不相关文档集合的大小。对于某个词语或单词 d_i 来说，假设包含这个词语的文档数量共有 n_i 个，而其中相关文档有 r_i 个，那么不相关文档中包含这个单词的文档数量则为 n_i-r_i。再考虑表中第 2 列，因为相关文档个数是 R，而其中出现过单词 d_i 的有 r_i 个，那么相关文档中没有出现过这个单词的文档个数为 $R-r_i$ 个，同理，不相关文档中没有出现过这个单词的文档个数为$(N-R)-(n_i-r_i)$个。从表中可以看出，如果我们假设已经知道 N、R、n_i、r_i 的话，其他参数可以靠这 4 个值推导出来。

采用最大似然估计，计算 $p_i=\frac{r_i}{R}$，$s_i=\frac{n_i-r_i}{N-R}$。为了避免 r_i 是 0 导致的 log0 无法计算的问题，采用相关文档和不相关文档都加 0.5 的平滑方法。这样得到 $p_i=\frac{r_i+0.5}{R+1}$，$s_i=\frac{n_i-r_i+0.5}{N-R+1}$。把这些值放入打分公式，得到：

$$\sum_{i:d_i=q_i=1} \log \frac{(r_i+0.5)/(R-r_i+0.5)}{(n_i-r_i+0.5)/(N-n-R+r_i+0.5)}$$

这个打分公式没有考虑词频，相关度比考虑词频的公式低 50%。

Okapi BM25（简称 BM25）是一种相关性排序函数，适用于搜索引擎根据与给定搜索查询的相关性对匹配文档进行排序。

1. 排名函数

BM25 是一个基于单词集合的检索函数，它根据出现在每个文档中的查询词对匹配文档集合排序，而不管查询词在文档内相互之间的联系。它不是一个单一的函数，而实际上是有略微不同的组件和参数变化的一群函数的集合。一个最典型的具体的函数如下。

假定有一个查询词组 Q，含有关键词 $q_1, q_2, ..., q_n$，用 BM25 给文档 D 评分的公式：

$$\mathrm{SCORE}(D,Q)=\sum_{i=1}^{n} \mathrm{IDF}(q_i)\cdot\frac{f(q_{i,}D)\cdot(k_1+1)}{f(q_{i,}D)+k_1\cdot(1-b+b\cdot\frac{|D|}{\mathrm{avgdl}})}$$

其中 $f(q_i, D)$是检索词 q_i 在文档 D 中的频率，$|D|$是文档 D 以单词为单位的长度，avgd1 是从抽取出的文档的文本集合的平均文档长度。k_1 和 b 是自由参数，通常选择 k_1= 2.0 和 b = 0.75。IDF(q_i)是检索词 q_i 的 IDF（文档频率倒数）权重。IDF(q_i)的通常计算公式：

$$\mathrm{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

这里，N 是集合中文档的总数，$n(q_i)$是包含 q_i 的文档个数。

5.1.1 使用 BM25 检索模型

基本上可以在索引设置中定义类似于自定义分析器的自定义 BM25 相似度，也就是定义 b 和 k_1 的值。例如：

```
 curl -XPUT "http://<server>/<index>" -d '
{
  "settings": {
    "similarity": {
      "custom_bm25": {
        "type": "BM25",
        "b":       0 ,
         "k1" : 0.9
      }
    }
  }'
```


5.1.2 参数调优

为了在 BM25 中调整这些参数，它对数据集非常依赖。简单的方法是：调整参数，然后检查结果，如果不满意，更改参数并再次测试结果。也可以使用像遗传算法或蚁群算法（Ant Colony Optimization）这样的启发式算法自动调参。

TF 归一化调优的经典方法是旋转归一化方法，但这个方法存在集合依赖问题。

可以下载 TREC 提供的数据集进行自动调参。数据集中的 Ad hoc Search 对一个固定的文档集合，根据用户输入的查询问题返回相关性降序输出的文档列表。

Jenetics（https://github.com/jenetics/jenetics）是一个 Java 编写的遗传算法库。它被设计成明确地分离算法的几个概念，如基因、染色体、基因型、表型、群体和适应度函数（Fitness Function）。

5.2 学习评分

PageRank 导致了 Google 炸弹。合并多个特征比一个特征更准确。PageRank 仅用于连接结构特征。

利用用户点击日志分析出哪些文档和查询词最相关。根据用户搜索行为调整搜索结果排序。此外，还可以通过社交网络判断文档相关度。

学习评分采用机器学习方法训练出采用多种特征的模型用来对文档相关度评分。它是一种有监督或半监督的机器学习问题，其目标是从训练数据自动构建评分模型。这样恶意用户更难以操作排名。有人说学习评分是一种超越 PageRank 的算法。

可以使用流行度数据用于训练，如访问量、用户在页面停留了多久。

5.2.1　基本原理

可以使用人工标注出一个理想的文档相关性排序。然后采用一种学习评分算法学习出模型。LambdaMART 是一种当前流行的学习评分算法，由微软的 Chris Burges 提出。目前，LambdaMART 在工业界被广泛使用，包括 Bing、Facebook 都在实际业务中使用了该算法。

LambdaMART 算法是从 Pairwise 方法逐渐发展起来的方法。Pairwise 方法的主要思想是将排序问题形式化为二元分类问题。Pairwise 方法通过考虑两两文档之间的相对相关度来进行排序。例如，文档 *X* 比文档 *Y* 更相关，还是更不相关，这是一个二元分类问题。其目标是最小化反转的排名，也就是让损失最小。RankNet 算法是一个 Pairwise 方法，它使用交叉熵作为损失函数。损失函数的值越低说明机器学得的当前排序越趋近于理想排序。RankNet 算法可以使用神经网络模型，也可以使用渐进梯度回归树（Gradient Boost Regression Tree，GDBT）模型。

如果 GDBT 模型求解过程使用求梯度的 Lambda 方法，就是 LambdaMART 算法。这里的 MART（Multiple Additive Regression Tree），也就是 GBDT（Gradient Boosting Decision Tree）。Lambda 的含义是一个待排序的文档下一次迭代应该排序的方向（向上或向下）和强度。

机器学习评分包 RankLib（https://sourceforge.net/p/lemur/wiki/RankLib/）用于 Lemur 搜索引擎，但也可以修改后和 Lucene 一起使用。

可以直接在命令行运行 RankLib.jar：

```
> java -jar RankLib.jar
```

5.2.2　准备数据

可以使用电影数据（https://github.com/holgerbrandl/themoviedbapi）。先在电影数据库网站（https://www.themoviedb.org）中申请 API。

然后从 https://jcenter.bintray.com/下载依赖的 jar 包。

如果使用 maven，那么可以手动将此 repo 添加到 pom.xml 或 settings.xml 中：

```
<repositories>
    <repository>
        <id>jcenter</id>
        <releases>
```

```
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>
```

搜索的代码如下。

```
TmdbApi tmdb = new TmdbApi(apiKey);
TmdbSearch search = tmdb.getSearch();
String keyWords = "Rambo";        //查询词
MovieResultsPage movieResultsPage = search.searchMovie(keyWords, null, null, false, null);
List<MovieDb> movies = movieResultsPage.getResults();
System.out.println(movies.size());
System.out.println(movies);        //按相关度输出电影列表
```


首先创建一个判断列表。传统上，用一个 0～4 的数值表示。0 表示不相关，4 表示完全相关。

考虑搜索“rambo”。如果“doc_1234”是电影“Rambo”，而“doc_5678”是“Turner and Hooch”，那么可以做出以下两个判断：

4,rambo,doc_1234 # "Rambo" 是搜索"rambo"的精确匹配（第 4 等级）；

0,rambo,doc_5678 # "Turner and Hooch"完全不相关（0 级）。

为了使用 Ranklib，需要做一些预处理来检查查询和文档，并生成一组定量特征，假设可能会预测相关性等级。这里的一个特征是测量查询，文档或查询和文档之间的关系的数值。可以任意决定，例如，特征 1 是查询关键字在电影标题中出现的次数，而特征 2 可能对应于电影概览字段中关键字出现的次数。

这种训练集的通用文件格式如下：

```
等级 qid:<queryId> 1:<feature1Val> 2:<feature2Val>... #评论
```

举个例子，当查看查询“rambo”时，将其称为查询 Id 1，注意以下特征值。

- 特征 1：Rambo 在电影“Rambo”的标题中出现 1 次，0 次在 Turner and Hootch。
- 特征 2：Rambo 在电影“Rambo”的概述领域发生 6 次，0 次在 Turner and Hootch。

然后上面的判断表就变成了：

```
4 qid：1 1：1 2：6
0 qid：1 1：0 2：0
```

qid:1 的相关电影具有比不相关的匹配更高的特征 1 和 2 的值。

一个完整的训练集用在成千上万或更多查询上的分级文档表示了这个想法：

```
4 qid:1 1:1 2:6
0 qid:1 1:0 2:0
4 qid:2 1:1 2:6
```

```
3 qid:2 1:1 2:6
0 qid:2 1:0 2:0
...
```

训练的目的是生成一个函数（这里也简单地称之为模型），它接受输入特征 1, 2,…, *n* 并输出相关性等级。下载 Ranklib 后，可以训练出一个模型文件，如下所示。

```
> java -jar bin/RankLib.jar -train train.txt -ranker 6 -save mymodel.txt
```

这个命令训练数据 train.txt 以生成 LambdaMART 模型（ranker 6），将模型的文本表示输出到 mymodel.txt。一旦有了一个好的模型，就可以用它作为排名函数来产生相关性分数了。

5.2.3　Elasticsearch 学习排名

elasticsearch-learning-to-rank（https://github.com/o19s/elasticsearch-learning-to-rank）以插件方式提供了让 Elasticsearch 支持学习排名的方法。例如，要在 Elasticsearch 5.4.0 上安装插件的 0.1.2 版，请执行以下操作。

```
./bin/elasticsearch-plugin install http://es-learn-to-rank.labs.o19s.com/ltr-query-0.1.2-es5.4.0.zip
```


让 ltr 查询使用该模型来打分。下面的“dummy”是打分过的模型。每个“特征”都会反映出在训练时使用的查询。

```
GET /foo/_search
{
    "query": {
        "ltr": {
        "model": {
            "stored": "dummy"
        },
        "features": [{
            "match": {
                "title": userSearchString
            }
        },{
                "constant_score": {
                    "query": {
                        "match_phrase": {
                            "overview": "userSearchString"
                        }
                    }
                }
            }]
    }
    }
}
```

因为 ltr 模型评估可能会花费较长时间，所以最好在 rescore 上下文中使用此查询。更实际的 ltr 的实现如下所示。

```
{
    "query": {/*基本查询在这里*/},
    "rescore": {
        "query": {
            "rescore_query": {
              "ltr": {
                  "model": {
                      "stored": "dummy"
                  },
                  "features": [{
                      "match": {
                          "title": userSearchString
                      }
                  },{
                      "constant_score": {
                          "query": {
                              "match_phrase": {
                                  "overview": "userSearchString"
                              }
                          }
                      }
                  }]
              }
            }
        }
    }
}
```


Elasticsearch 查询的相关性得分使用了大量的特征。例如，与相关性相关的一个特征可能是用户的搜索关键字具有强大的标题 TF * IDF 相关性分数。

```
{
    "query": {
        "match": {
            "title": userSearchString
        }
    }
}
```

另一个有希望的特征可能是概述短语匹配，也许忽略 TF * IDF 得分并坚持用一个 constant_score。

```
{
    "query": {
```

```
            "constant_score": {
                "query": {
                    "match_phrase": {
                        "overview": userSearchString
                    }
                }
            }
        }
    }
```

5.3 本章小结

本章介绍了 Lucene 全文索引库的基本使用方法和常用的定制修改方法，介绍了索引文件的格式，以及通过分发索引文件到其他服务器来实现分布式搜索。为了更好地实现搜索准确性，可以改进检索模型。

多词查询假设如果一篇文档包含一些这样的句子，至少两个查询词在同一句话中出现，则这篇文档更相关。也就是说，多个查询词之间要有更短的距离，如“Pisa Tower”。如果词出现的位置之间的距离增加，则底层意思可能已经变了。

BM25 源自 20 世纪八九十年代伦敦城市大学第一个实现这个函数的系统——Okapi 信息检索系统。它是基于 20 世纪七八十年代由 Stephen E. Robertson, Karen Spärck Jones 等人开发的概率检索框架。从 20 世纪 80 年代末开始，概率模型（特别是以 Okapi 系统为代表的 BM25 系列算法）出现并逐渐分享了经典模型在信息检索模型领域的地位，成为新兴的且功能强大、表现越来越出色的模型。

BM25 和 BM25F 两种模型在 TREC 文本检索评测会议中都有优越于其他的表现并且公认是目前 IR 范围内最为先进的检索模型。BM25 适用于没有结构的全文检索，而 BM25F 适用于结构化的文档检索，也就是用于有好几个全文搜索列的情况。

在 RankNet 模型的基础上改进出了 LambdaRank 模型。LambdaRank 模型上又发展出了 LambdaMART 机器学习评分检索模型。

除了用于检索文档，机器学习评分算法还可以用于推荐引擎。例如，购物网站给访问网站的用户推荐商品。

除了 RankLib，还可以使用 XGBoost（https://github.com/dmlc/xgboost）实现学习评分。

在第 6 章，我们将介绍索引库在用户界面中的调用方法。

搜索引擎用户界面

搜索相关页面主要包括首页和搜索结果页。如果用户输入搜索词是空，则可以显示一个对信息分类导航的页面。首页主要包含搜索条区域，此外可以包括一些推荐信息，以及当前热门信息。

对于互联网搜索来说，搜索结果界面往往采用 JSP 或 ASP.NET、PHP、Python 等技术来实现。搜索联想词的页面效果可以用 Ajax 来实现。

这里首先介绍采用 JSP 和自定义标签实现的搜索界面，然后介绍 REST 架构的搜索界面。

6.1 JSP 实现搜索界面

为了方便开发和部署到 Tomcat 等 Web 应用服务器，建议在 MyEclipse 中开发搜索页面。

为了防止报 Bad version number in .class file 错误，需要 MyEclipse 和 Tomcat 统一使用同一个 JDK。当安装 All in One 版本的 MyEclipse 时，就会自带它的 JDK。建议将 MyEclipse 中的 JDK 路径修改为自己的 JDK，具体修改方法如下。

（1）选择 Window→Preferences→Java→Installed JREs。

（2）选择 Add...→在弹出的对话框中单击 Browse...按钮→选择你的 JDK 路径（如 C:\Program Files\Java\jdk1.6.0_10）→单击 OK 按钮。

（3）在刚配置的 JDK 路径上打钩，单击 OK 按钮设置为默认 JDK 路径。

JSP 只是输出一个字符串，然后交给 Web 服务器，最后由用户的浏览器显示网页。Tomcat 是一个常用的 Web 服务器。为了避免 Tomcat 输出的网页乱码，一般使用 utf-8 编码。为了正确接收 HTTP 参数，在 JSP 页面中设置：

```
request.setCharacterEncoding("utf-8");
```

为了对其他的搜索引擎友好（SEO），搜索关键词通过 GET 取得，而不要通过 POST 取得查询参数。翻页参数也最好通过 GET 方式传递。同样为了 SEO，搜索词会出现在搜索结果页的标题中。查询关键词的参数一般用 q 或 query 表示。

org.apache.commons.lang3.StringUtils 中的 defaultIfEmpty 方法返回空值的替代值。

```
String query = StringUtils.defaultIfEmpty(request.getParameter("query"),"");
```

为了防止 JavaScript 脚本注入，需要使用 StringEscapeUtils.escapeHtml 对用户查询转码。

首先在控制台测试搜索功能，然后再考虑在 Web 环境中加载索引库。

6.1.1　用于显示搜索结果的自定义标签

为了实现代码复用，定义一个专用于根据关键词查询返回结果的 Taglib。搜索结果页是一个表格型的数据。Listlib 实现了对数据的封装和抽象，可以通过它来控制显示的结果数量，如可以指定每页显示 20 条记录或 10 条记录。实际执行 Lucene 搜索的类继承 ListCreator 接口，并把搜索结果通过 ListContainer 类的实例返回即可。Listlib 中定义的 tag 如下。

1．init

listlib 的起始 tag。创建一个 ListCreator 对象，运行该对象的 execute() 方法，并把它存储在 HttpServletRequest 属性中。这是一个容器 tag，所以在 jsp 页面使用时，其他的 tag 都必须嵌套在这个 tag 中间。

它的主要属性有：通过 name 指定一个名字，因为需要通过这个名字来把 ListCreator 对象存储在 HttpServletRequest 属性。通过 listCreator 来指定创建 ListCreator 的对象。通过 max 来声明每页必须显示的记录条数。

2．hasResults

如果 list 有结果，就会执行这个 tag，否则会跳过。

3．hasNoResults

如果 list 没有结果就会执行这个 tag，否则会跳过。

4．prop

返回 list 中的属性值。和搜索结果总体相关的信息可以通过它来显示，如搜索提示词、搜索结果分类统计等。

5．hasPrev

如果还可以继续往回遍历，就会显示这个 tag 中的内容，否则就跳过。

6．hasNext

如果还可以继续向下遍历，就会显示这个 tag 中的内容，否则就跳过。

7．iterate

遍历 ListContainer 中的元素。

8．iterateProp

从 Iterator 的当前对象返回属性，如返回标题通过<list:iterateProp property="title"/>。

在 listlib.tld 文件中定义 Taglib。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
        "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>listlist Taglib</short-name>
    <uri></uri>
    <description>listlib Taglib</description>

    <tag>
      <name>init</name>
      <tag-class>com.bitmechanic.listlib.InitTag</tag-class>

    <tei-class>com.bitmechanic.listlib.InitTagExtraInfo</tei-class>
    <body-content>JSP</body-content>

      <description></description>

      <attribute>
          <name>name</name>
          <required>true</required>
          <rtexprvalue>true</rtexprvalue>
      </attribute>

      <attribute>
          <name>class</name>
          <required>false</required>
          <rtexprvalue>true</rtexprvalue>
      </attribute>

      <attribute>
          <name>listCreator</name>
          <required>false</required>
          <rtexprvalue>true</rtexprvalue>
      </attribute>

      <attribute>
          <name>max</name>
          <required>false</required>
          <rtexprvalue>true</rtexprvalue>
      </attribute>
```

```
</tag>

<tag>
    <name>hasResults</name>
    <tag-class>com.bitmechanic.listlib.HasResultsTag</tag-class>
    <description></description>

    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>hasSuggest</name>
    <tag-class>com.bitmechanic.listlib.HasSuggestTag</tag-class>
    <description></description>

    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>hasNoResults</name>
    <tag-class>com.bitmechanic.listlib.HasNoResultsTag</tag-class>
    <description></description>

    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>prop</name>
    <tag-class>com.bitmechanic.listlib.PropTag</tag-class>
    <description></description>
```

```
    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

    <attribute>
        <name>property</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>hasPrev</name>
    <tag-class>com.bitmechanic.listlib.HasPrevTag</tag-class>
    <description></description>

    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>hasNext</name>
    <tag-class>com.bitmechanic.listlib.HasNextTag</tag-class>
    <description></description>

    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>prevLink</name>
    <tag-class>com.bitmechanic.listlib.PrevLinkTag</tag-class>
    <description></description>

    <attribute>
```

```
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>nextLink</name>
    <tag-class>com.bitmechanic.listlib.NextLinkTag</tag-class>
    <description></description>

    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>iterate</name>
    <tag-class>com.bitmechanic.listlib.IterateTag</tag-class>
    <description></description>

    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>iterateProp</name>
    <tag-class>com.bitmechanic.listlib.IteratePropTag</tag-class>
    <description></description>

    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

    <attribute>
        <name>property</name>
```

```
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>

    </tag>

    <tag>
        <name>iterateURLEncodeProp</name>
        <tag-class>com.bitmechanic.listlib.IteratePropURLEncodeTag</tag-class>
        <description></description>

        <attribute>
            <name>name</name>
            <required>false</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>

        <attribute>
            <name>charset</name>
            <required>false</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>

        <attribute>
            <name>property</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>

    </tag>

</taglib>
```


将中文字符作为参数时，需要用对应字符集编码这个字符串。这里为 URL 编码专门写了一个自定义标签 IteratePropURLEncodeTag。

使用 IterateURLEncodeProp 的例子如下。

```
<a href=
"folder.jsp?folder=<list:iterateURLEncodeProp property="folder"/>&docType=<list:iterateProp property="docType"/>"
```

6.1.2 使用 Listlib

执行搜索的 Java Bean。

```
public class SearchWeb implements ListCreator {
```

```
        private String _query;

        private Client server;   //在 Web 容器内全局唯一
        private static Logger logger = Logger.getLogger(SearchWeb.class.getName());

        //只调用一次
        public void init(String host) throws Exception {
            server = new PreBuiltTransportClient(Settings.EMPTY)
            .addTransportAddress(new InetSocketTransportAddress(InetAddress
                    .getByName(host), 9200));
        }
    }
```

在开发搜索 Web 界面之前，首先写一个控制台方式运行的搜索程序测试一下索引库。使用存根类专门测试 ListCreator 的实现类 SearchWeb：

```
    String host = "localhost"; //Es 服务地址

    String query = "地质"; //查询词

    SearchWeb search = new SearchWeb();

    search.init(host);
    search.setQuery(query);
    //翻页参数
    int offset = 0;
    int max = 10;

    PageContextImpl context = new PageContextImpl(); //存根类
    context.request = new ServletRequestImpl();

    ListContainer lc = search.execute(context,   offset, max);
    lc.setUrl("Search.jsp");
    System.out.println("result size:" + lc.getSize());   //输出结果总数
    Iterator it = lc.getIterator();
    while (it.hasNext()) {
        HashMap<String, String> row = (HashMap<String, String>) it.next();
        System.out.println("url:" + row.get("url"));
        System.out.println("title:" + row.get("title"));
        System.out.println("body:" + row.get("body"));
    }
```

在 JSP 调用实现类之前。需要区分哪些类是静态的，是全局唯一的；哪些对象需要在页面内即时创建和使用；哪些对象在整个用户会话期间内有效。

jsp:useBean 标签创建一个 Bean 实例并指定它的名字和作用范围。

```
    <!—定义全局唯一的搜索 Bean – 它实现了 ListCreator 的 execute 方法 -->
```

```
<jsp:useBean id="searchInf" class="com.lietu.search.SearchWeb" scope="application">
<!—指定 ES 服务器的 IP 地址-->
<% searchInf.init("localhost"); %>
</jsp:useBean>
<!—把查询从 http 的 get 参数设置到搜索对象中去-->
<jsp:setProperty name="searchInf" property="query" value="<%=query%>"/>
<!—执行搜索并把返回结果封装到 ListContainer -->
<list:init name="information"    listCreator="searchInf" max="20">
```

6.1.3 实现翻页

翻页链接需要指定相对路径。<base>标签为所有链接指定相对路径。首先通过 Java 代码取得相对路径。

```
<%
String path = request.getContextPath();
String basePath = request.getScheme()
        +"://"+request.getServerName()
        +":"+request.getServerPort()+path+"/";
%>
```

然后在<base>标签中指定相对路径是 basePath：

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <base href="<%=basePath%>">
  <title>旅游活动搜索</title>
</head>
```

需要在分页器的构造方法中告诉分页器符合查询条件的结果共有多少条。分页器则告诉查询对象，从第几条结果开始返回。另外一个预先固定设置好的值是每页最多显示的结果数。

推荐使用翻页组件 Pager-taglib（https://github.com/cnliuy/pager-taglib）。Pager-taglib 是一个 JSP 标签库，支持多种风格的分页显示。为在 JSP 上显示分页信息而设计了一套标签，通过这些标签的不同组合，会形成多种不一样的分页页面，风格各异，它自带的 DEMO 就有 7 种左右的分页风格，包括 Google 的分页风格。而需要定制自己的风格的分页页面也非常简单。使用 Pager-taglib 的流程如下。

（1）复制 pager-taglib.jar 包到 lib 目录下，不需要修改 web.xml。

（2）在 JSP 页面中使用 taglib 指令引入 pager-taglib 标签库。

（3）使用 pager-taglib 标签库进行分页处理。

通过 maxPageItems 参数设定每页最多显示的结果数。在 JSP 页面中使用翻页标签库的例子如下。

```
<pg:pager       url="SearchAction.action"
            items="<%=Integer.parseInt(listSize)%>"
```

```
            maxPageItems="20"
            maxIndexPages="10"
            export="currentPageNumber=pageNumber"
            scope="request">
  <pg:param name="query" value="<%=query%>"/>
…
</pg:pager>
```

其中在 pg:pager 标签中定义了 action 的 URL 地址，在 pg:param 标签中定义了查询参数 query。

假设按 10 条记录分页，显示第一页的实现如下。

```
QueryParser parser = new QueryParser(defaultField,
                                     analyzer);

Query query = parser.parse(queryString);
//最多返回 10 个文档，也就是返回和查询词最匹配的前 10 个文档
TopDocs hits = searcher.search(query, 10);
System.out.println("返回结果总条数："+hits.totalHits);
for (int i = 0; i < hits.scoreDocs.length; i++) {
    Document hitDoc = searcher.doc(hits.scoreDocs[i].doc);
    System.out.println("第"+i+"条："+hitDoc.get("title"));
}
if (hits.totalHits > hits.scoreDocs.length){
    System.out.println("还有更多结果在后面排队");
}else{
    System.out.println("结果显示完毕");
}
```

返回从 offset 开始的 rows 行的实现如下。

```
searcher.search(query,offset+rows);  //最多返回查询的前(offset+rows)条
System.out.println("返回结果总条数："+hits.totalHits);
for (int i = offset; i < hits.scoreDocs.length; i++) {
    Document hitDoc = searcher.doc(hits.scoreDocs[i].doc);
    System.out.println("第"+i+"条："+hitDoc.get("title"));
}
```

pager-taglib 在输出的页面中生成链接 search.jsp?query=%E7%9A%84&pager.offset=10，其中包含开始位置的参数。在 InitTag 类中得到开始返回结果的位置。

```
public static final String OFFSET_KEY = "pager.offset";

public int doStartTag() throws JspException {
        String offsetStr = pageContext.getRequest().getParameter(OFFSET_KEY);
        int offset = Integer.parseInt(offsetStr);
}
```

6.2 使用 Spring 实现的搜索界面

为了实现微服务架构，可以结合 Spring 框架封装搜索请求，展现搜索结果。前端通过表格显示搜索结果的 Javascrip 插件有 jqGrid、jQuery EasyUI 等。

6.2.1 实现 REST 搜索界面

要构建 REST API，必须了解以下四件事情。

（1）控制器：控制器控制 HTTP 请求与应用程序逻辑之间的交互。

（2）资源：指定连接到 Es 服务器的参数。

（3）链接：翻页链接。

（4）如何构建这些链接：在 REST 控制器中使用 spring-data-common 的 PagedResources Assembler，它可以在响应中生成下一页/上一页的链接。

首先，@Configuration 注解是基于 Java 的 Spring 配置使用的主要构件，它本身是使用 @Component 进行元注解的，它使得被注解类标准的 bean 也是组件扫描的候选项。@Configuration 类的主要目的是作为 Spring IoC 容器的 bean 定义源。

注解成配置的类不应该是 final 的，这个类应该有一个没有参数的构造函数。

```
@Configuration
@ComponentScan( basePackages = "org.rest" )
@PropertySource({
   "classpath:rest.properties",
   "classpath:web.properties"
})
public class AppConfig{

    @Bean
    public static PropertySourcesPlaceholderConfigurer
       properties() {

        return new PropertySourcesPlaceholderConfigurer();
    }
}
```

使用 HttpMessageConverter 和注解创建 RESTful 服务。

```
@Configuration
@EnableWebMvc
public class WebConfig{
    //
}
```

@EnableWebMvc 注解会在类路径中检测到 Jackson 和 JAXB 2 的存在，并自动创建和

注册默认的 JSON 和 XML 转换器。

测试：

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration(
   classes = { ApplicationConfig.class, PersistenceConfig.class },
   loader = AnnotationConfigContextLoader.class )
public class SpringTest {

    @Test
    public void whenSpringContextIsInstantiated_thenNoExceptions(){
        //When
    }
}
```

Java 配置类仅用@ContextConfiguration 注解指定，新的 AnnotationConfigContextLoader 从@Configuration 类加载 bean 定义。

请注意，这个测试并没有包含 WebConfig 配置类，因为它需要在 Servlet 上下文中运行。这里并未提供。

@Controller 是 RESTful API 整个 Web 层中的中心构件，例子代码如下。

```
@Controller
@RequestMapping("/foos")
class FooController {

    @Autowired
    private IFooService service;

    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    public List<Foo> findAll() {
        return service.findAll();
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    @ResponseBody
    public Foo findOne(@PathVariable("id") Long id) {
        return RestPreconditions.checkFound( service.findOne( id ));
    }

    @RequestMapping(method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    @ResponseBody
    public Long create(@RequestBody Foo resource) {
        Preconditions.checkNotNull(resource);
        return service.create(resource);
```

```
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.OK)
    public void update(@PathVariable( "id" ) Long id, @RequestBody Foo resource) {
        Preconditions.checkNotNull(resource);
        RestPreconditions.checkNotNull(service.getById( resource.getId()));
        service.update(resource);
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.OK)
    public void delete(@PathVariable("id") Long id) {
        service.deleteById(id);
    }

}
```


控制器的实现类是非公开的，因为它不需要公开。

通常，控制器是依赖关系链中的最后一个——它接收来自 Spring 前端控制器（DispathcerServlet）的 HTTP 请求，并将它们转发到服务层。如果没有使用场景通过直接引用来注入或操纵控制器，那么就没有必要将它声明为公开的类。

请求映射是直接的，与任何控制器一样，映射的实际值及 HTTP 方法都用于确定请求的目标方法。@RequestBody 会将方法的参数绑定到 HTTP 请求的正文，而@ResponseBody 对响应和返回类型执行相同的操作。

它们还确保使用正确的 HTTP 转换器对资源进行编组和解组。将进行内容协商，以选择要使用哪个活跃的转换器，主要基于 Accept 头，尽管也可以使用其他 HTTP 头来确定表示。

来自 Spring Boot 的@RestController 注释基本上是一个快捷方式，可以帮助我们避免必须定义@ResponseBody。

6.2.2 REST API 中的 HTTP PUT

当进行部分更新时，可以使用 HTTP PATCH。

首先定义实现 REST API 以更新具有多个字段的 HeavyResource：

```
public class HeavyResource {
    private Integer id;
    private String name;
    private String address;
    //...
```

首先，我们需要创建使用 PUT 来处理资源的完整更新的端点：

```
@PutMapping("/heavyresource/{id}")
public ResponseEntity<?> saveResource(@RequestBody HeavyResource
```

```
                                                    heavyResource,
                    @PathVariable("id") String id) {
        heavyResourceRepository.save(heavyResource, id);
        return ResponseEntity.ok("resource saved");
    }
```

这是更新资源的标准端点。

现在，我们通常会由客户端更新地址字段。在这种情况下，我们不想将所有的字段都传给整个 HeavyResource 对象，但我们希望通过 PATCH 方法只能更新地址字段。可以创建一个 HeavyResourceAddressOnly DTO 来表示地址字段的部分更新。

接下来，可以利用 PATCH 方法发送部分更新：

```
    @PatchMapping("/heavyresource/{id}")
    public ResponseEntity<?> partialUpdateName(
            @RequestBody HeavyResourceAddressOnly partialUpdate,
            @PathVariable("id") String id) {

        heavyResourceRepository.save(partialUpdate, id);
        return ResponseEntity.ok("resource address updated");
    }
```


使用这种更细粒度的 DTO 可以发送需要更新的字段，而不用发送整个 HeavyResource 的开销。

如果我们有大量的这样的部分更新操作，也可以跳过为每个外部创建一个定制的 DTO，而仅使用一个 Map：

```
    @RequestMapping(value = "/heavyresource/{id}", method = RequestMethod.PATCH, consumes =
MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<?> partialUpdateGeneric(
      @RequestBody Map<String, Object> updates,
      @PathVariable("id") String id) {

        heavyResourceRepository.save(updates, id);
        return ResponseEntity.ok("resource updated");
    }
```

该解决方案将为我们提供更多的灵活性来实现 API。然而，我们也失去了一些像验证这样的东西。

最后，我们为这两个 HTTP 方法编写测试。首先，我们要通过 PUT 方法测试完整资源的更新：

```
    mockMvc.perform(put("/heavyresource/1")
      .contentType(MediaType.APPLICATION_JSON_VALUE)
      .content(objectMapper.writeValueAsString(
        new HeavyResource(1, "Tom", "Jackson", 12, "heaven street")))
      ).andExpect(status().isOk());
```

通过使用 PATCH 方法来实现部分更新：

```
mockMvc.perform(patch("/heavyrecource/1")
   .contentType(MediaType.APPLICATION_JSON_VALUE)
   .content(objectMapper.writeValueAsString(
      new HeavyResourceAddressOnly(1, "5th avenue")))
   ).andExpect(status().isOk());
```

还可以为更通用的方法编写一个测试代码：

```
HashMap<String, Object> updates = new HashMap<>();
updates.put("address", "5th avenue");

mockMvc.perform(patch("/heavyresource/1")
      .contentType(MediaType.APPLICATION_JSON_VALUE)
      .content(objectMapper.writeValueAsString(updates))
   ).andExpect(status().isOk());
```


6.2.3 Spring-data-elasticsearch

Spring Data 提供了一套数据访问层（DAO）的解决方案，致力于减少数据访问层的开发量。Spring Data Elasticsearch 是 Spring Data 的子项目。实现了 Spring Data 访问 Elasticsearch 存储，并提供了 Spring Data JPA（Java 持久化 API）模型的访问方式。这个项目的地址是：

https://github.com/spring-projects/spring-data-elasticsearch。

Spring Data 使用一个叫作 Repository 的接口类为基础，Repository 定义如下。

```
public interface Repository<T, ID extends Serializable> {
}
```

Repository 是访问底层数据模型的超级接口。而对于某种具体的数据访问操作，则在其子接口中定义。例如，Spring Data Elasticsearch 项目中定义了访问 Elasticsearch 中数据的 ElasticsearchRepository 接口。ElasticsearchRepository 扩展了 PagingAndSortingRepository，它为分页和排序提供了内置的支持。

所有继承 Repository 接口的界面都由 Spring 管理，此接口作为标识接口，功能是控制领域模型。Spring Data 可以让我们只定义接口，只要遵循 Spring Data 的规范，就无须写实现类。Spring 可以根据接口中定义的方法名实现 Repository。

引入包：

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-elasticsearch</artifactId>
    <version>${spring-data-elasticsearch-version}</version>
    <exclusions>
        <exclusion>
```

```
                    <artifactId>elasticsearch</artifactId>
                    <groupId>org.elasticsearch</groupId>
                </exclusion>
            </exclusions>
        </dependency>
```

在 XML 文件中配置连接参数：

```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
           xsi:schemaLocation="http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd
            http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

        <elasticsearch:transport-client id="client"
            cluster-nodes="127.0.0.1:9300" cluster-name="lietu" />
        <bean name="elasticsearchTemplate" class="org.springframework.data.elasticsearch.core. ElasticsearchTemplate">
            <constructor-arg name="client" ref="client"/>
        </bean>

        <elasticsearch:repositories base-package="org.springframework.data.elasticsearch.repositories" />

    </beans>
```

为自定义方法扩展 ElasticsearchRepository：

```
    public interface BookRepository extends Repository<Book, String> {

        List<Book> findByNameAndPrice(String name, Integer price);

        List<Book> findByNameOrPrice(String name, Integer price);

        Page<Book> findByName(String name,Pageable page);

        Page<Book> findByNameNot(String name,Pageable page);

        Page<Book> findByPriceBetween(int price,Pageable page);

        Page<Book> findByNameLike(String name,Pageable page);

        @Query("{\"bool\" : {\"must\" : {\"term\" : {\"message\" : \"?0\"}}}}")
        Page<Book> findByMessage(String message, Pageable pageable);
    }
```

使用 Repository 索引单个文档：

```
@Autowired
private SampleElasticsearchRepository repository;

String documentId = "123456";
SampleEntity sampleEntity = new SampleEntity();
sampleEntity.setId(documentId);
sampleEntity.setMessage("some message");

repository.save(sampleEntity);
```

使用 Repository 索引多个文档（批量索引）：

```
@Autowired
private SampleElasticsearchRepository repository;

String documentId = "123456";
SampleEntity sampleEntity1 = new SampleEntity();
sampleEntity1.setId(documentId);
sampleEntity1.setMessage("some message");

String documentId2 = "123457"
SampleEntity sampleEntity2 = new SampleEntity();
sampleEntity2.setId(documentId2);
sampleEntity2.setMessage("test message");

List<SampleEntity> sampleEntities = Arrays.asList(sampleEntity1, sampleEntity2);

//批量索引
repository.save(sampleEntities);
```


ElasticsearchTemplate 是 Elasticsearch 操作的核心支持类。使用 ElasticsearchTemplate 索引单个文档：

```
String documentId = "123456";
SampleEntity sampleEntity = new SampleEntity();
sampleEntity.setId(documentId);
sampleEntity.setMessage("some message");
IndexQuery indexQuery =
  new IndexQueryBuilder().withId(sampleEntity.getId()).withObject(sampleEntity).build();
elasticsearchTemplate.index(indexQuery);
```

使用 ElasticsearchTemplate 索引多个文档（批量索引）：

```
@Autowired
private ElasticsearchTemplate elasticsearchTemplate;
```

```
List<IndexQuery> indexQueries = new ArrayList<IndexQuery>();
//第 1 个文档
String documentId = "123456";
SampleEntity sampleEntity1 = new SampleEntity();
sampleEntity1.setId(documentId);
sampleEntity1.setMessage("some message");

IndexQuery indexQuery1 =
        new IndexQueryBuilder().withId(sampleEntity1.getId())
            .withObject(sampleEntity1).build();
indexQueries.add(indexQuery1);

//第 2 个文档
String documentId2 = "123457";
SampleEntity sampleEntity2 = new SampleEntity();
sampleEntity2.setId(documentId2);
sampleEntity2.setMessage("some message");

IndexQuery indexQuery2 =
      new IndexQueryBuilder().withId(sampleEntity2.getId())
            .withObject(sampleEntity2).build()
indexQueries.add(indexQuery2);

//批量索引
elasticsearchTemplate.bulkIndex(indexQueries);
```


使用 Elasticsearch Template 搜索实体：

```
@Autowired
private ElasticsearchTemplate elasticsearchTemplate;

SearchQuery searchQuery = new NativeSearchQueryBuilder()
        .withQuery(queryString(documentId).field("id"))
        .build();
Page<SampleEntity> sampleEntities =
      elasticsearchTemplate.queryForPage(searchQuery,SampleEntity.class);
```

可以使用实体类定义索引结构：

```
import org.springframework.data.annotation.Id;
import org.springframework.data.annotation.Version;
import org.springframework.data.elasticsearch.annotations.Document;
import org.springframework.data.elasticsearch.annotations.Field;
import org.springframework.data.elasticsearch.annotations.FieldType;

@Document(indexName = "book",type = "book" , shards = 1, replicas = 0, indexStoreType = "memory",
refreshInterval = "-1")
```

```
public class Book {

    @Id
    private String id;
    private String name;
    private Long price;
    @Version
    private Long version;

    public Map<Integer, Collection<String>> getBuckets() {
        return buckets;
    }

    public void setBuckets(Map<Integer, Collection<String>> buckets) {
        this.buckets = buckets;
    }

    @Field(type = FieldType.Nested)
    private Map<Integer, Collection<String>> buckets = new HashMap();

    public Book(){}

    public Book(String id, String name,Long version) {
        this.id = id;
        this.name = name;
        this.version = version;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Long getPrice() {
        return price;
```

```
        }

        public void setPrice(Long price) {
            this.price = price;
        }

        public long getVersion() {
            return version;
        }

        public void setVersion(long version) {
            this.version = version;
        }
    }
```

实际执行索引结构定义：

```
    elasticsearchTemplate.putMapping(Book.class);
```

可以通过 xml 配置设置存储库。

使用节点客户端连接到服务器：

```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
           xsi:schemaLocation="http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch.xsd
            http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

        <elasticsearch:node-client id="client" local="true"/>

        <bean name="elasticsearchTemplate" class="org.springframework.data. elasticsearch.core.Elasticsearch
Template">
            <constructor-arg name="client" ref="client"/>
        </bean>

    </beans>
```

使用 Transport 客户端连接到服务器：

```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
           xsi:schemaLocation="http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch.xsd
```

```
                http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

        <elasticsearch:repositories base-package="com.xyz.acme"/>

        <elasticsearch:transport-client id="client" cluster-nodes="ip:9300,ip:9300" cluster- name=
"elasticsearch" />

        <bean  name="elasticsearchTemplate"  class="org.springframework.data.elasticsearch.core.Elasticsearch
Template">
            <constructor-arg name="client" ref="client"/>
        </bean>

    </beans>
```

6.2.4 Spring HATEOAS

HATEOAS（The Hypermedia As The Engine Of Application Statue）是 REST 架构的主要约束。Spring-HATEOAS（https://github.com/spring-projects/spring-hateoas/）项目是一个 Spring MVC 扩展，可帮助我们构建 HATEOAS 样式的 API。

在 Spring HATEOAS 项目中，不需要查找 Servlet 上下文，也不需要将路径变量连接到基本 URI。

Spring HATEOAS 提供了创建 URI 的三个类：ResourceSupport、Link 和 ControllerLinkBuilder。这些类用于创建元数据并将其与资源表示相关联。

资源是一个包含基本属性的组件：字符串、数字等类型，可以通过链接访问相关资源并改变当前资源的状态。

每个资源必须至少有一个叫作 self 的链接，指向当前资源本身。可以将资源视为实体，因为它具有属性并可以与其他实体相关联。资源就是我们的模式（Model），当对应 MVC 术语时。在 Spring-HATEOAS 中，可以从 ResourceSupport 类扩展资源。Resource Support 已经实现了对添加链接的支持，所以我们的工作只是关心属性以及从哪里构建这些 链接。

如果资源仅由属性链接组成，则应有一个简单的方法来构建它们。使用 Spring-HATEOAS，可以使用任何 LinkBuilder 实现构建链接。但是，建议使用 ControllerLinkBuilder，它可以仅使用 Java DSL 构建链接。链接将指向控制器及其方法，如果需要，可以重构这些方法，链接构造代码也将被重构。

可以使用 ControllerLinkBuilder.linkTo 方法创建一个新的链接实例，然后决定要使用哪种形式的构建来构造。如果决定使用指向 Method 参数的 linkTo，则可以使用 ControllerLinkBuilder.methodOn 方法，这将帮助我们构建一个直接指向当前资源的公共路径的链接。链接构造如下所示。

```
    linkTo(methodOn(ControllerClass.class).someMethod(parameter)).withSelfRel();
```

增加一个自身链接的代码：

```
import static org.sfw.hateoas.mvc.ControllerLinkBuilder.*;

Link link = linkTo(YourController.class).slash(resource.getName()).withSelfRel();
resource.add(link);
```

6.3 实现搜索接口

本节介绍从基本的布尔逻辑查询开始，到指定范围的查询，以及搜索结果排序等实现方法。

6.3.1 编码识别

搜索引擎的查询关键词是很重要的一个参数，这个参数是一个查询字符串的 URL 编码。一个非 ASCII 字符的 URL 编码由一个“%”符号后面跟着两个十六进制的数字组成。中文搜索需要判断传入的这个字符串的 URL 编码是 GBK 还是 UTF-8 格式。

符合 J2EE 标准的 Web 服务器（如 Tomcat）通过调用 request.getQueryString() 方法可以得到原始提交的参数。比如发送：

```
http://localhost/search.do?query=%B0%A1
```

getQueryString 方法得到的字符串是：

```
query=%B0%A1
```

然后调用编码识别方法，用正确的编码来解码。

```
String input = "%E6%B5%B7%E6%8A%A5%E7%BD%91";
String codingName=getEncoding(input);//判断编码
System.out.println(URLDecoder.decode(input, codingName)); //用正确的编码来解码
```

主要的开发工作是根据输入字符串判断编码。

GB2312 的字符编码范围在%B0%A1～%F7%FE 之间，如表 6-1 所示。

表 6-1　汉字编码对照表

字　　符	编　　码
啊	%B0%A1
阿	%B0%A2
鞍	%B0%B0
齾	%F7%FE

汉字 Unicode 编码范围为\u4e00～\u9fa5。UTF-8 汉字 URL 编码后的取值范围为：

```
%E4%B8%80～%E4%BF%BF
%E5%B8%80～%E5%BF%BF
```

```
%E6%B8%80～%E6%BF%BF
%E7%80%80～%E7%BF%BF
%E8%80%80～%E8%BF%BF
%E9%80%80～%E9%BE%A5
```

像左括号和右括号这样的 ASCII 编码小于 128 的字符编码都小于%80，如左括号字符编码是%28，右括号字符编码是%29，而所有的汉字编码，无论是 UTF-8 或 GBK，每个字节的编码都是大于或等于%80 的。

```
//判断是否可能是 UTF-8 编码的汉字
public static boolean isUtf8(String code1,String code2,String code3) {
    if (code1.compareTo("E4") >= 0 && code1.compareTo("E9") <= 0 &&
            code2.compareTo("80") >= 0 && code2.compareTo("BF") <= 0 &&
            code3.compareTo("80")>=0 &&code3.compareTo("BF")<=0) {
        return true;
    }
    return false;
}
//判断是否可能是 GB2312 编码的汉字
public static boolean isGb2312(String code1,String code2) {
    if (code1.compareTo("B0") >= 0 && code1.compareTo("F7") <= 0 &&
            code2.compareTo("A0")>=0 &&code2.compareTo("FF")<=0) {
        return true;
    }
    return false;
}
//根据字符列表猜测字符编码
public static String getEncodeByList(List<String> code) {
    if(code.size() >= 2 && code.size()%2 == 1 && code.size()%3 == 0) {
        return "utf8";
    }
    else if(code.size() >= 2 && code.size()%2 == 0 && code.size()%3 != 0) {
        return "gbk";
    }
    else if(code.size()%6 == 0) {
        for(int m=0;m<code.size();m = m+6) {
            if( ! isUtf8(code.get(m), code.get(m+1), code.get(m+2)) &&
               isGbk(code.get(m), code.get(m+1)) &&
               isGbk(code.get(m+2), code.get(m+3)) ) {
                return "gbk";
            } else if(isUtf8(code.get(m), code.get(m+1), code.get(m+2)) &&
                    ! isGbk(code.get(m), code.get(m+1)) ) {
                return "utf8";
            }
            if(! isUtf8(code.get(m+3), code.get(m+4), code.get(m+5)) &&
               isGbk(code.get(m+2), code.get(m+3)) &&
```

```
                    isGbk(code.get(m+4), code.get(m+5)) ) {
                        return "gbk";
                    } else if(isUtf8(code.get(m+3), code.get(m+4), code.get(m+5)) &&
                        !isGbk(code.get(m+2), code.get(m+3))) {
                        return "utf8";
                    }
                }
            }
            return "utf8";
        }
```

根据有限状态机的思想把字符串切分成数组。首先定义状态类：

```
public enum CharType {
    Enter,//碰到%
    Code1,//碰到%后的第 1 个字符
    Code2,//碰到%后的第 2 个字符
}
```


然后根据上一个状态以及当前的字符决定下一个状态。进入下一个状态时，有可能执行判断字符编码的动作。

```
public static String getURLEncoding(String url) {
    List<String> codes = new ArrayList<String>();
    CharType currentSate = null;
    char c1='\0';
    char c2='\0';
    for(int i=0; i<url.length(); ++i) {
        char currentChar = url.charAt(i);
        if(currentChar == '%') {
            if(currentSate == CharType.Code2 )      {
                char[] s1 = {c1,c2};
                codes.add(new String(s1));
            }
            currentSate = CharType.Enter;
        }else if(currentSate==CharType.Enter) {
            c1 = currentChar;
            currentSate = CharType.Code1;
        }else if(currentSate==CharType.Code1)      {
            c2 = currentChar;
            currentSate = CharType.Code2;
        }else if(currentSate==CharType.Code2)      {
            char[] s1 = {c1,c2};
            codes.add(new String(s1));
            currentSate = null;
            return getEncodeByList(codes);
```

```
            }
        }
        if(currentSate==CharType.Code2) {
            char[] s1 = {c1,c2};
            codes.add(new String(s1));
        }
        return getEncodeByList(codes);
    }
```

6.3.2 布尔搜索

字词混合搜索：

```
static void addShould(BoolQueryBuilder qb, PageContext context,String argName,
        String singleField,String wordField) {
    String qString = context.getRequest().getParameter(argName);
    if (StringUtils.isNotEmpty(qString)) {
        MatchPhraseQueryBuilder singleQB =
            QueryBuilders.matchPhraseQuery(singleField, qString);   //字查询
        MatchPhraseQueryBuilder wordQB =
            QueryBuilders.matchPhraseQuery(wordField, qString);   //词查询

        QueryBuilder currentQB =
            QueryBuilders.boolQuery().should(singleQB).should(wordQB);
        qb.should(currentQB);
    }
}
```

用布尔查询来合并多个查询条件：

```
BoolQueryBuilder qb = QueryBuilders.boolQuery();

addShould(qb, context, "repname","repnameS","repname");
addShould(qb, context, "repeditor","repeditorS","repeditor");
addShould(qb, context, "repsubunit");
addShould(qb, context, "repsubdate");
```

6.3.3 搜索结果排序

可以按单列或多列排序，但需要保证排序的列是不做切分处理的，也就是对该列做索引时设置 Field.Index.NOT_ANALYZED。例如，“url”网址列没有做过切分，可以按该列排序，而标题列“title”做过切分，不能按该列排序。

经常需要按日期倒序排序，为了支持对日期列排序，需要把日期转换成统一的字符串格式“yyyyMMddHHmmssSSS”。如果精度低，则字符串长度相应变短。

索引日期的例子：

```
Date pubDate = rs.getDate("pubDate");
Field f = new Field(“pubDate”,
        DateTools.dateToString(pubDate, DateTools.Resolution.DAY),//精度到天
        Field.Store.YES,
        Field.Index.NOT_ANALYZED);
```

按日期倒序排序的例子：

```
Sort sort= new Sort(new SortField("pubDate",SortField.STRING,true));
ScoreDoc[] hits = searcher.search(query,null,1000,sort).scoreDocs;
```

也可以对多个字段排序，如先按地区列 area 排序，然后按类别 type 排序：

```
Sort sort= new Sort(new SortField[]{new SortField("area"),new SortField("type")});
ScoreDoc[] hits = searcher.search(query,null,1000,sort).scoreDocs;
```

还可以通过 SortComparatorSource 自定义排序方法。

6.4　实现相似文档搜索

有时需要检索与给定文档（如 BBS 讨论区内某一帖子）相似的文档。打开一个新闻网页，往往在下面有块区域，显示和这篇新闻相关的新闻。对一个卖商品的网站来说，当顾客正在浏览一件商品时，如果能把和这件商品性能、作用相近的商品也同时罗列在网页的左边，万一顾客想要的商品正好就在其中，那么这个网站的营业额肯定会有所提高。

找出和指定电影相关的电影：

```
{
    "more_like_this" : {
        "fields" : ["title", "description"],
        "docs" : [
        {
            "_index" : "imdb",
            "_type" : "movies",
            "_id" : "1"
        },
        {
            "_index" : "imdb",
            "_type" : "movies",
            "_id" : "2"
        }],
        "min_term_freq" : 1,
        "max_query_terms" : 12
    }
}
```

实现原理是：构造一个 MoreLikeThis(MLT)的对象 MLT，然后调用 mlt.like(1)，这里的

1 是 Lucene 内部的文档编号。然后搜索一下，取前几个结果就是与此文档最为相似的。

like(int docNum)方法返回的 Query 是怎么产生的呢？首先根据传入的 docNum 找出该文档里去除停用词后的高频词，然后用这些高频词生成 Queue，最后把 Queue 传进 search 方法得到最后的结果。其主要思想就是认为这些高频词足以表示文档信息，然后通过搜索得到最后与此 doc 类似的结果。

另外一个简单的用法是：要求提供与提供的文本类似的文档。

```
GET /_search
{
    "query": {
        "more_like_this" : {
            "fields" : ["title", "description"],
            "like" : "Once upon a time",
            "min_term_freq" : 1,
            "max_query_terms" : 12
        }
    }
}
```


在进行 MLT 之前，实际将最小词频和最小文档频率应用于输入。如果输入文本中只有一个“apple”，则不符合 MLT 的限制，因为最小词频设置为 2，如果将输入更改为“apple apple”就能起作用了。

```
POST /test_index/_search
{
    "query": {
        "more_like_this": {
            "fields": [
                "text"
            ],
            "like_text": "apple apple",
            "min_term_freq": 2,
            "percent_terms_to_match": 1,
            "min_doc_freq": 1
        }
    }
}
```

Java API 调用 MLT：

```
String[] fields = { "title", "description" };
String[] likeTexts = { "Once upon a time" };
Item[] likeItems = {new MoreLikeThisQueryBuilder.Item()};
MoreLikeThisQueryBuilder queryBuilder = new MoreLikeThisQueryBuilder(
        fields, likeTexts, likeItems);
```

6.5　实现 AJAX 搜索联想词

搜索输入框中的下拉提示给用户一个有参考意义的搜索词表，有时还提供用户搜索该词预期的结果数量。这个功能有时也称自动完成（AutoCompleter），一般是由浏览器端的 AJAX 代码完成的。

搜索词表可以从用户搜索日志中统计出来，搜索次数多的词排在前面。除了按一般的设计，即为每个用户提供统一的词表，还可以对每个用户提供个性化的推荐词表。例如，用户输入“汽”时，“汽车”的搜索次数比“汽油发电机”多，所以排在提示词列表的前面。如果搜索日志比较少，无法挖掘出足够多的推荐搜索词，可以考虑从文本中挖掘一些关键词作为推荐搜索词。因为后台索引一般都在不断变化，推荐搜索词右侧显示的“**结果”并不是实时搜索出的结果，只是一个估计值，只具有参考价值。为了实现搜索提示效果，需要用到词典的前缀匹配。

6.5.1　估计查询词的文档频率

为了对于用户输入任何词都可以显示一个估计的搜索结果数量，需要计算这个词的文档频率。例如，用户输入“NBA 直播”，可以根据“NBA”和“直播”的文档频率估计“NBA 直播”的文档频率。假设“NBA”和“直播”之间的出现没有依赖关系，则可以简化计算如下：

$$P(\text{"NBA"} \cap \text{"直播"}) = P(\text{"NBA"}) \cdot P(\text{"直播"})$$

这里的 $P(\text{"NBA"} \cap \text{"直播"})$ 是联合概率，可以认为是“NBA 直播”的出现概率，而 $P(\text{"NBA"})$ 和 $P(\text{"直播"})$ 是每个词出现的概率。假设索引中的总文档数量是 N，而 $P(\text{"NBA"}) = \text{Freq}(\text{"NBA"})/N$，$P(\text{"直播"}) = \text{Freq}(\text{"直播"})/N$。

因此“NBA 直播”的文档频率为：

$$\text{Freq}(\text{"NBA直播"}) = N \times \frac{\text{Freq}(\text{"NBA"})}{N} \times \frac{\text{Freq}(\text{"直播"})}{N}$$

而 Freq("NBA")和 Freq("直播")所代表的文档频率在 Lucene 中可以通过 org.apache.lucene.search.Searcher 的 docFreq(Term term) 方法得到。因为多个词之间并不一定满足独立出现的假设，因此这个估计值有可能偏低。

6.5.2　搜索联想词总体结构

当用户在浏览器的输入框中输入查询词时，JavaScript 代码捕获用户即时输入的数据并向服务器发送请求。自动完成功能的总体结构如图 6-1 所示。

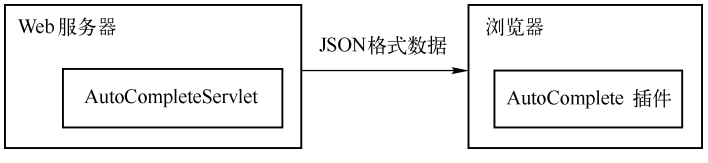

图 6-1　自动完成功能总体结构

如果使用 struts2，则不要让 struts2 的 FilterDispatcher 把所有的 URL 都拦截了。

```
<!-- 定义 struts2 的 FilterDispathcer 的 Filter -->
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    </filter>

<!-- FilterDispatcher 用来初始化 struts2 并处理.action 和.jsp 的 Web 请求。 -->
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>*.action</url-pattern>
    </filter-mapping>
```

6.5.3 服务器端处理

在用户输入一个搜索字的同时由 Web 服务器中的一个 Servelet(AutoCompleteServlet)从后台取数，可以直接在内存中管理查询，而不是访问数据库取数。先设计词典格式。它由 3 列组成：第 1 列是词，第 2 列是搜索返回结果数量，第 3 列是用户搜索次数，中间用%隔开。例如：

综合教程第一册%34%2

搜索词是“综合教程第一册”，搜索返回结果数量是 34，用户搜索了 2 次。这样把用户搜索次数多的关键词放在前面优先显示。我们构造一个在“词典查找算法”部分已经介绍过的“Trie”树词典来实现快速前缀匹配查找：

```
/**
 * 返回以一个前缀开始的所有关键词的数组
 *
 *@param   prefix          前缀
 *@param   numReturnValues 返回数组的最大长度
 *@return                  返回数组结果
 */
public TSTItem[] matchPrefix(String prefix, int numReturnValues) {
    TSTNode startNode = getNode(prefix);
    if (startNode == null) {
        return null;
    }
    ArrayList<TSTItem> sortKeysResult = new ArrayList<TSTItem>();

    ArrayList<TSTItem> wordTable = sortKeysRecursion(
            startNode.EQKID,
            ((numReturnValues < 0) ? -1 : numReturnValues),
            sortKeysResult);
    int retNum = Math.min(numReturnValues,wordTable.size());
```

```
        Select.selectRandom(wordTable,wordTable.size(),retNum,0);
        TSTItem[] fullResults = new TSTItem[retNum];
        for(int i=0;i<retNum;++i)       {
            fullResults[i] = wordTable.get(i);
        }

        return fullResults;
    }

    /**
     * 按顺序返回关键词，包括当前节点和与当前节点相关的所有节点对应的关键词
     * 关键词将按顺序追加到结果的尾数
     *@param    currentNode             当前节点
     *@param    sortKeysNumReturnValues     最多返回结果数
    *@param    sortKeysResult2                 到目前为止的结果
     *@return     一个列表
    */
    private ArrayList<TSTItem> sortKeysRecursion(
        TSTNode currentNode,
        int sortKeysNumReturnValues,
        ArrayList<TSTItem> sortKeysResult2) {

        if (currentNode == null) {
            return sortKeysResult2;
        }

        ArrayList<TSTItem> sortKeysResult =
            sortKeysRecursion(
                currentNode.LOKID,
                sortKeysNumReturnValues,
                sortKeysResult2);

        if (currentNode.data != 0) {
            sortKeysResult.add(
                    new TSTItem(getKey(currentNode),
                            currentNode.data,
                            currentNode.weight)
                    );
        }

        sortKeysResult =
            sortKeysRecursion(
                currentNode.EQKID,
                sortKeysNumReturnValues,
                sortKeysResult);
```

```
        return sortKeysRecursion(
                currentNode.HIKID,
                sortKeysNumReturnValues,
                sortKeysResult);
    }
```

可以写一个简单的测试代码：

```
    public static void main(String[] args) {
        SuggestDic sugDic =    SuggestDic.getInstance();
        String prefix = "m";
        TSTItem[] ret = sugDic.matchPrefix(prefix, 10);
        for(TSTItem i:ret ) {
            System.out.println(i.key+":"+i.data+":"+i.weight);
        }
    }
```


服务器传递给客户端的 JSON 数据格式是一个数组，例如：

```
    ["lietu","lucene"]
```

因为 JSON 格式是一个比较简单的数据传输格式，所以采用了 JSON.org 提供的一个简单的生成包。

```
    JSONArray jsonarray = new JSONArray();
    jsonarray.put("lietu");
    jsonarray.put("lucene");
    System.out.println(jsonarray.toString());
```

根据对象生成 JSON 串的 POJO 类：

```
    public class SuggestItem {
        public String w; //词
        public int c; //结果数量

        public int getC() {
            return c;
        }
        public void setC(int c) {
            this.c = c;
        }
        public String getW() {
            return w;
        }
        public void setW(String w) {
            this.w = w;
        }
    }
```

通过 Servlet 输出 JSON 时，需要设置正确的 MIME 类型（application/json）和字符编码。假定服务器使用 UTF-8 编码，则可以使用以下代码输出编码后的 JSON 文本：

```
response.setContentType("application/json;charset=UTF-8");
response.setCharacterEncoding("UTF-8");
```

这样自动完成的 Servlet 类可以写成：

```
String val = request.getParameter("q");

String message = null;
try {
      SuggestDicByWord sugDic = SuggestDicByWord.getInstance();
      SuggestItem[] items = sugDic.matchPrefix(val, 10);

      JSONValue lMyValue = JSONMapper.toJSON(items);
      message = Escape.toUnicodeEscapeString(lMyValue.render(false));
      message = lMyValue.render(false);
} catch (Exception e) {
      e.printStackTrace();
}

response.setContentType("application/json; charset=utf-8");
PrintWriter out = response.getWriter();
if (message != null) {
      out.println(message);
} else {
      out.println("");
}
```


把 AutoCompleteServlet 通过 web.xml 部署到 URL 地址“/autoComplete”。

```
<servlet>
   <servlet-name>AutoCompleteServlet</servlet-name>
   <servlet-class>com.lietu.autocomplete.AutoCompleteServlet</servlet-class>
</servlet>
   <servlet-mapping>
   <servlet-name>AutoCompleteServlet</servlet-name>
   <url-pattern>/autoComplete</url-pattern>
</servlet-mapping>
```

服务器端修改成：

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
      throws IOException {
            String val = request.getParameter("q");
            String message = null;
```

```
        SuggestItem[] items = null;
        try {
            SuggestDic sugDic =    SuggestDic.getInstance();

            items = sugDic.matchPrefix(val, 10);//查找 Trie 树

            JSONValue lMyValue = JSONMapper.toJSON(items);
            message = Escape.toUnicodeEscapeString( lMyValue.render(false) );

        } catch (Exception e) {
            e.printStackTrace();
        }

        response.setContentType("text/html; charset=utf-8");
        PrintWriter out = response.getWriter();
        if(message!=null)              {
            out.println(message);
        } else {
            out.println("");
        }
}
```


可以通过 EasyMock 测试这个 Servlet 的返回值：

```
String queryWord = "P";
//录制 mock 对象
HttpServletRequest request = createMock(HttpServletRequest.class);
HttpServletResponse response = createMock(HttpServletResponse.class);
ServletConfig servletConfig = createMock(ServletConfig.class);
ServletContext servletContext = createMock(ServletContext.class);

AutoCompleteServlet instance = new AutoCompleteServlet();

//初始化 servlet 一般由容器承担，用 servletConfig 作为参数初始化，此处模拟容器行为
instance.init(servletConfig);
//在某些方法被调用时设置期望的返回值
//如下这样就不会去实际调用 servletConfig 的 getServletContext 方法，而是直接返回
//servletContext，由于 servletConfig 是 mock 出来的，所以可以完全控制
expect(servletConfig.getServletContext()).andReturn(servletContext).anyTimes();

expect(request.getParameter("q")).andReturn(queryWord);

PrintWriter pw=new PrintWriter(System.out,true);
expect(response.getWriter()).andReturn(pw).anyTimes();
response.setContentType("text/html; charset=utf-8");

//重放 mock 对象
```

```
replay(request);
replay(response);
replay(servletConfig);
replay(servletContext);

instance.doPost(request, response);

pw.flush();

//检查预期和实际结果
verify(request);
verify(response);
verify(servletConfig);
verify(servletContext);
```

返回一个 JSON 数组格式的数据，如果要支持中文，还要对汉字编码。

6.5.4　浏览器端处理

剩下的就是在前台通过 AJAX 组件库 jQuery 中的 Autocomplete 插件（http://jqueryui.com/demos/autocomplete/）来完成显示了。

先到官方网站下载 jQuery 的最新版本，然后需要 jQuery UI 的 3 个核心组件：Core、Widget、Position，还有 AutoComplete 插件。

将插件中的 JavaScript 文件和 css 文件分别置于 Web 项目中的 js 文件夹和 css 文件夹中。最后将这些文件导入到需要搜索联想词的页面，一般是搜索首页和搜索结果页面，也就是网页的头信息中包含这些文件。

```
<head>
    <link rel="stylesheet" href="css/jquery.ui.all.css">
    <script language="javascript" src="js/jquery-1.5.1.js"></script>
    <script language="javascript" src="js/jquery.ui.core.js"></script>
    <script language="javascript" src="js/jquery.ui.widget.js"></script>
    <script language="javascript" src="js/jquery.ui.position.js"></script>
    <script language="javascript" src="js/jquery.ui.autocomplete.js"></script>
</head>
```

AutoComplete 插件与一个 HTML 的 Input 标签相结合。

现在网页中加一个输入标签：

```
<input id="query" autocomplete="off" />
```

注意，对输入标签来说需要增加 autocomplete="off"。如果不加，浏览器可能不会提交 HTTP 请求到后台。

当 DOM（文档对象模型）已经加载，并且页面（包括图像）已经完全呈现时，会发生 ready 事件。在 jQuery 中，使用$(function)定义 ready 事件的处理函数。

在 ready 事件中增加 autocomplete 控件：

```
<script>
$(function() {
    var availableTags = ["ActionScript", "AppleScript", "Asp"];
    $( "#query" ).autocomplete({
        source: availableTags
    });
});
</script>
```

通过 AJAX 方式取得数据，访问后台 URL 地址位于“./autoComplete”的数据源的代码：

```
<script>
$(function() {
    $( "#query" ).autocomplete({
        source: "./autoComplete",
        minLength: 2
    });
});
</script>
```


它自动传递一个称为 term 的参数，这个参数中包括 query 输入框中的值。例如，当用户在搜索框输入 o 这个字母时，浏览器就会发送下面这个请求给服务器：

```
HTTP GET autoComplete?term=o
```

服务器传递给客户端的 JSON 数据格式是一个搜索词组成的数组，例如：

```
["open office","online backup","opera","onkyo",...]
```

用户选择词后，一般直接跳转到这个词的搜索结果，而不是只在输入框中显示这个搜索词，之后用户需要再次按搜索按钮才返回搜索结果。有时需要支持用户选择提示词直接跳转到搜索结果页面。在 JavaScript 中，通过 location.href 实现跳转。用户选择提示词后直接搜索的实现如下。

```
$(function() {
    $("#query").autocomplete({
        source: "./autoComplete",
        minLength: 2,
        select: function( event, ui ) {
            window.location.href="./searchAction.action?query=" + ui.item.value;
        }
    });
});
```

在这里，通过 ui.item.value 得到用户选择的搜索词。

如果不能正常工作，首先看浏览器是否已经发送 HTTP 请求，然后再看 Servlet 返回的

结果。为了方便跟踪错误，可以使用 FireFox 中的 Firebug 插件调试网页中的 JavaScript 代码。在 Firebug 中，可以为 JavaScript 设置断点，可以暂停执行 JavaScript 并看到每个变量的当前值。如果代码速度慢，还可以通过 JavaScript 配置器看性能，快速发现性能瓶颈。

6.5.5　拼音提示

为了支持汉语拼音感应，需要把所有的词生成出拼音列。Trie 树可以看成关键词和值的映射。拼音列和词本身都可以作为关键词，值这一列则存放词原型。例如，对于“厦门”这个词，会存储两个关键词和值的映射。

xiamen -> 厦门

厦门 -> 厦门

这样当用户输入“厦”或“xia”都可能提示“厦门”这个词。

对于基本的中文词提示来说，关键词和值都是一样的。另外，注音程序把中文词转换成拼音，这部分数据支持汉语拼音感应功能。

因为存在多音字，按词注音会有好的结果。可以在 Trie 树的值域中存储一个词对应的拼音。


```
public static String yin(String sentence){//传入一个字符串作为要处理的对象
    int senLen = sentence.length();//首先计算传入的这句话的字符长度
    int i = 0;//用来控制匹配的起始位置的变量
    StringBuilder result = new StringBuilder(senLen);
    TernarySearchTrie.MatchRet matchRet = new TernarySearchTrie.MatchRet("",0);
    while (i < senLen){//  如果 i 小于此句话的长度就进入循环
        boolean match = dic.matchLong(sentence, i, matchRet);//正向最大长度匹配
        if (match){//已经匹配上，按词注音
            i = matchRet.end;
            result.append(matchRet.data);
        } else//如果没有找到匹配上的词，就按单字注音
        {
            result.append(ziYin.zi2Yin(sentence.charAt(i)));
            ++i; //下次匹配点在这个字符之后
        }
    }
    return result.toString();
}
```

6.5.6　部署总结

提示词词典 suggestDic.txt 可以放在 WEB-INF/classes/dic/ 路径。AutoCompleteServlet 可以放在 WEB-INF/lib/路径，通过 web.xml 发布。界面用到的 JavaScript 脚本 jquery.js、jquery.ajaxQueue.js 、jquery.autocomplete.css 和 jquery.autocomplete.js 可以放在 ROOT/js 路径。

可以给用户提供个性化的提示词。同样输入“大”字，对于影迷，提示“大话西游”，对于美食爱好者提示“大福”（一种日式甜品）。

6.5.7 Suggester

Suggester 基本的运作原理是将输入的文本分解为 token，然后在索引的字典里查找相似的 term 并返回。根据使用场景的不同，Elasticsearch 里设计了 4 种类别的 Suggester，分别是 Term Suggester、Phrase Suggester、Completion Suggester 和 Context Suggester。

Term Suggester 只基于 analyze 的单个词去提供建议，并不会考虑多个词之间的关系。API 调用方只需为每个 token 挑选 options 里的词，组合在一起返回给用户前端即可。

Phrase Suggester 在 Term Suggester 的基础上，会考量多个词之间的关系，比如是否同时出现在索引的原文里，相邻程度，以及词频等。

Completion Suggester 使用有限状态转换（FST）查找提示词。FST 会被 Es 整个装载到内存里，进行前缀查找速度极快，但 FST 只能用于前缀查找。

Completion Suggester 考虑索引中的所有文档，但通常希望通过某些标准来筛选和（或）提升提示词。例如，想要提示根据品牌过滤商品，或者想根据歌曲的风格提升歌曲名称。

为了实现建议过滤和（或）提升，可以在配置完成字段时添加上下文映射，可以为自动完成字段定义多个上下文映射。每个上下文映射都有唯一的名称和类型。有文本类别和地理两种类型。上下文映射在字段映射的 contexts 参数下进行配置。

以下定义类型中每个类型具有自动完成字段的一个上下文映射：

```
PUT place
{
    "mappings": {
        "shops" : {
            "properties" : {
                "suggest" : {
                    "type" : "completion",
                    "contexts": [
                        {
                            "name": "place_type",
                            "type": "category"
                        }
                    ]
                }
            }
        }
    }
}
```

定义名为 place_type 的类别上下文，从 cat 字段读取类别。

```
PUT place_path_category
```

```
{
    "mappings": {
        "shops" : {
            "properties" : {
                "suggest" : {
                    "type" : "completion",
                    "contexts": [
                        {
                            "name": "place_type",
                            "type": "category",
                            "path": "cat"
                        }
                    ]
                }
            }
        }
    }
}
```


通过类别上下文，可以将一个或多个类别与索引时的提示词相关联。在查询时，可以通过相关类别过滤和提升提示词。

如果定义了路径（path），那么从文档中的路径中读取类别，否则必须在提示字段中送出，代码如下所示。

```
PUT place/shops/1
{
    "suggest": {
        "input": ["timmy's", "starbucks", "dunkin donuts"],
        "contexts": {
            "place_type": ["cafe", "food"]
        }
    }
}
```

6.6 推荐搜索词

搜索引擎中往往有个可选搜索词的列表，当搜索结果太少时，可以帮助用户扩展搜索内容，或者当搜索结果过多时，可以帮助用户深入定向搜索。这称为查询推荐（Query Suggestion）。

一种方法是从搜索日志中挖掘字面相似的词作为相关搜索词列表。从一个给定的词语挖掘多个相关搜索词，可以用编辑距离为主的方法查找一个词的字面相似词，如果候选的相关搜索词很多，就要筛选出最相关的 10 个词。还可以从查询日志中聚类出一些相关的查询。

有的搜索词是有歧义的，如搜索“李娜”，相关搜索词应该提示：歌手、网球运动员，

还是跳水运动员。

6.6.1 挖掘相关搜索词

下面是利用 Lucene 筛选给定词的最相关词的方法。

```
private static final String TEXT_FIELD = "text";
/**
        * @param words  候选相关词列表
        * @param word  要找相关搜索词的种子词
        * @return
        * @throws IOException
        * @throws ParseException
*/
static String[] filterRelated(HashSet<String> words, String word) {
            StringBuilder sb = new StringBuilder();
            for(int i=0;i<word.length();++i)       {
                sb.append(word.charAt(i));
                sb.append(" ");
            }
            RAMDirectory store = new RAMDirectory();
            //按字生成索引和查找，也可以按细粒度的词分开
            IndexWriter writer = new IndexWriter(store, new StandardAnalyzer(), true);
            for(String text:words) {
                Document document = new Document();
                Field textField = new Field(TEXT_FIELD, text,
                                        Field.Store.YES, Field.Index.TOKENIZED);
                document.add(textField);
                writer.addDocument(document);
            }
            writer.close();
            IndexSearcher searcher = new IndexSearcher(store);
            QueryParser queryParser = new QueryParser(TEXT_FIELD,
                                                    new StandardAnalyzer());
            Query query = queryParser.parse(sb.toString());
            Hits hits = searcher.search(query);
            int maxRet = Math.min(10, hits.length());
            String[] relatedWords = new String[maxRet];
            for (int i = 0; i < maxRet ; i++) {
               Document document = hits.doc(i);
               String text = document.get(TEXT_FIELD);
               System.out.println(text);
               relatedWords[i]=text;
```

```
            }
            searcher.close();
            store.close();
            return relatedWords;
    }
```

上述代码整理出这样的相关词表，第一列是关键词，后续是 10 个以内的相关搜索词：

```
集福轩婚礼%集福轩
手机定位跟踪系统%手机定位系统%手机定位%手机定位仪器
喷绘材料卖店电话%我要喷绘材料卖店电话
厦门房产%厦门租房%厦门新闻%厦门桑拿%房产%青岛房产%厦门%恒雄房产
送水果%送水%水果
三星传真机%三星手机
```

另一种方法是，可以把多个用户共同查询的词看成相关搜索词，这需要有记录用户 IP 的搜索日志才能实现。

可以通过 RelatedEngine 类查找某个关键词的相关词。

```
RelatedEngine re =new RelatedEngine(
                    new File("D:/lg/work/xiaoxishu/dic/relatedwords.txt"));
String word = "徐家汇";
String[] relatedWords = re.getRelated(word);
for(String w : relatedWords) {
    System.out.println(w);
}
```

输出如下：

```
上海徐家汇
徐汇
徐家汇价格是
上房徐家汇路附近有吗
```

当我们需要为新建立的搜索引擎开发相关搜索时，如果没有搜索日志而用户文本很多时，可以：

（1）首先运行 IndexMaker 从待搜索的文档中提取关键词并生成索引。

（2）然后运行 RelatedWords 从索引生成相关词表。

另外，还可考虑用日志记录用户对相关搜索词的选择。如果用户也选择了这个词，那么搜索词肯定和这个选择词相关了。

隐含语义索引（Latent Semantic Indexing，LSI）的原理是在相同的上下文中的词有相似的含义。http://code.google.com/p/airhead-research/是 Java 版本的实现。

Word2Vec Java 版本的实现（https://github.com/medallia/Word2VecJava）可以把词表示成向量，然后找出和这个词语义相近的词，并给出相似度。

6.6.2 使用多线程计算相关搜索词

如果要计算任意两个查询词之间的相关性，则发现相关搜索词的时间复杂度是 $O(n^2)$。例如，要分析 10 多 MB 的相关搜索词的用时，则单线程的计算量可能长达 24 小时。

我们使用 Java 中自带的轻量级线程池来实现数据分析。JDK1.5 以后的版本提供了一个轻量级线程池 ThreadPool。可以使用线程池执行一组任务，最简单的任务为不返回值给主调线程。要返回值的任务可以实现 Callable<T>接口，线程池执行任务并通过 Future<T>的实例获取返回值。

实现 Callable 方法的任务类的主要实现：

```
public class FindSimCall implements Callable<String[]> {
    private HashSet<String> words;        //总的搜索词集合
    private String s;                     //待发现相关词的词

    public FindSimCall(HashSet<String> w ,String source)    {
    words = w;
    s = source;
    }

    @Override
    public String[] call() throws Exception {
        System.out.println(s);
        //形成 related words 列表
        // …
        return relatedWords;
    }
}
```

主线程类的实现：

```
int threads = 4;
ExecutorService es = Executors.newFixedThreadPool(threads);

Set<Future<String[]>> set = new HashSet<Future<String[]>>();

for (final String s : words) {
    FindSimCall task = new FindSimCall(words,s);
    Future<String[]> future = es.submit(task);
    set.add(future);
}

FileOutputStream fos = new FileOutputStream(relatedWordsFile);
OutputStreamWriter osw = new OutputStreamWriter(fos,"GBK");
BufferedWriter writer = new BufferedWriter(osw);
```

```
for (Future<String[]> future : set) {
    String[] ret = future.get();

    for(String word:ret) {
        writer.write("%"+word);
    }
    writer.write( "\r\n" );
}
writer.close();
```

采用线程池可以充分利用多核 CPU 的计算能力，并且简化了多线程的实现。

6.7 查询意图理解

首先对用户查询预处理，如果无结果，再调整用户查询词来返回用户可能想要的文档。

6.7.1 拼音搜索

当用户输入一个全拼或拼音的简写，都可以搜索到数据。例如，用户输入“大衣”，“dayi”、“dy”、“day”都能搜索到。

可以通过查询扩展，把拼音转成中文搜索词。例如，dayi 转换成搜索词“大衣”。

拼音串作为键建立 Trie 树，如图 6-2 所示。

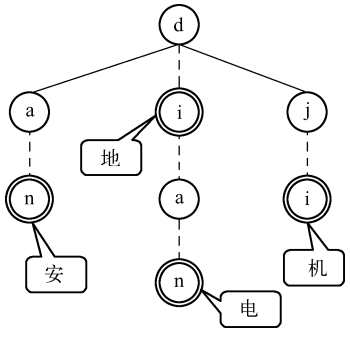

图 6-2　拼音建立的 Trie 树

也可以把拼音存到索引，做索引时加拼音。

Pinyin Analysis for Elasticsearch（https://github.com/medcl/elasticsearch-analysis-pinyin）能给词加注拼音。

6.7.2 无结果处理

用户输入“天梭的我表”。输入没有返回结果。系统提示： 根据“天梭表”、“我的”找到的结果。

拿着这几个字去索引里面查相关的词。根据“天\梭\表”三个字找到“天梭表”，根据“的”和“我”两个字找到“我的”。

6.8 集成其他功能

搜索引擎用户界面的一些功能还有：对用户输入的拼写纠错提示，搜索结果的分类统计，根据用户搜索词返回相关搜索词，在搜索结果中再次查找，记录和统计搜索日志。

6.8.1 拼写检查

当用户输入查询词“devloping distibutd saerch engies”，提示“developing distributed search engines”。这个功能叫作“Did you Mean”。如果搜索返回结果的数量小于阈值或匹配第一个结果的分值小于最小值就查找提示词。

Elasticsearch 中的 Term Suggester 可以实现英文拼写纠错。


```
{
  "my-suggestion": {
    "text": "davi",
    "term": {
      "field": "name_not_analyzed",
       "size" : 10
    }
  }
}
```

会返回类似这样的结果：

```
"options": [
                {
                    "text": "dave",
                    "score": 0.8333333,
                    "freq": 11
                },
                {
                    "text": "david",
                    "score": 0.6666666,
                    "freq": 6
                }
      ]
```

假设索引的名称是 rule，正在搜索的文本是 plabel，查询列是 pzInsKey。调用 Term Suggester 的 Java 代码如下。

```
SuggestBuilder.SuggestionBuilder suggestBuilder =
            new TermSuggestionBuilder("my-suggestion")
                .text("plabel").field("pzInsKey").suggestMode("always");
```

```
SuggestRequestBuilder requestBuilder = client.prepareSuggest("rule")
                                        .addSuggestion(suggestBuilder);
```

如果返回结果数量很少，可以直接把提示词的搜索结果放在原查询词返回结果的下面。

6.8.2　分类统计

首先定义分类统计信息类：

```
public class CatInf implements Comparable<CatInf> {
    public String name;       //分类名
    public int count;         //类别数量

    public int compareTo(CatInf obj){
                return (int)(obj.count - this.count);
    }
}
```


索引文档：

```
IndexQuery article1 =     new ArticleEntityBuilder("1")
            .title("article four").subject("computing")
           .addAuthor(RIZWAN_IDREES).addAuthor(ARTUR_KONCZAK).addAuthor(MOHSIN_HUSEN).
addAuthor(JONATHAN_YAN).score(10).buildIndex();
IndexQuery article2 =     new ArticleEntityBuilder("2")
            .title("article three").subject("computing")
            .addAuthor(RIZWAN_IDREES).addAuthor(ARTUR_KONCZAK).addAuthor(MOHSIN_HUSEN
).addPublishedYear(YEAR_2000).score(20).buildIndex();
IndexQuery article3 = new ArticleEntityBuilder("3")
    .title("article two").subject("computing")
     .addAuthor(RIZWAN_IDREES).addAuthor(ARTUR_KONCZAK).addPublishedYear(YEAR_2001).
addPublishedYear(YEAR_2000).score(30).buildIndex();
IndexQuery article4 = new ArticleEntityBuilder("4")
        .title("article one").subject("accounting")
        .addAuthor(RIZWAN_IDREES).addPublishedYear(YEAR_2002).addPublishedYear(YEAR_2001).
addPublishedYear(YEAR_2000).score(40).buildIndex();

elasticsearchTemplate.index(article1);
elasticsearchTemplate.index(article2);
elasticsearchTemplate.index(article3);
elasticsearchTemplate.index(article4);
elasticsearchTemplate.refresh(ArticleEntity.class, true);
```

对书按主题分类统计：

```
SearchQuery searchQuery = new NativeSearchQueryBuilder()
        .withQuery(matchAllQuery()).withSearchType(COUNT)
        .withIndices("articles").withTypes("article")
        .addAggregation(AggregationBuilders.terms("subjects").field("subject")).build();

Aggregations aggregations = elasticsearchTemplate.query(searchQuery,
        new ResultsExtractor<Aggregations>() {
            @Override
            public Aggregations extract(SearchResponse response) {
                return response.getAggregations();
            }
        });

Aggregation subjects = aggregations.asMap().get("subjects");
```

为了保证分类统计和查询的结果一致性，需要共用一个查询对象。实现分类统计的方法声明如下。


```
ArrayList<CatInf> factedCounter(IndexSearcher searcher, Query q)
```

二级子树展开的效果如图 6-3 所示。

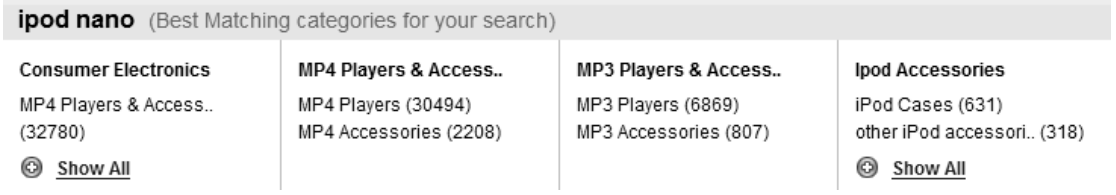

图 6-3　二级子树的展开效果

节点类定义如下。

```
public class CatNode {
    public int no;    //编码
    public String name; //节点名
    public boolean isLeaf;    //是否页节点
    public List<CatNode> children = null; //孩子节点
    public CatNode parent;    //父节点
    public int level;    //级别

    public CatNode(int no, String name,CatNode parentNo,int l, boolean isLeaf)    {
        this.no = no;
        this.name = name;
        this.parent = parentNo;
        this.level = l;
        this.isLeaf = isLeaf;
        if (!isLeaf)    {
            children = new ArrayList<CatNode>(5);
```

```
        }
    }

    public void addChildren(CatNode node) throws Exception    {
        if (isLeaf)    {
            throw new Exception("add child error to leaf node:" + no);
        }
        children.add(node);
    }

    public String toString()   {
        String temp = this.name;
        for (CatNode child : children)   {
            temp += "\n"+ "child:"+ child.no +":"+child.name ;
        }
        temp += "\n";
        return temp;
    }
}
```


用于查找父子节点的映射表：

```
public class CategoryMap extends HashMap<Integer, CatNode> {
    public CategoryMap(){ //构造方法
        String sql =
          "SELECT ID,isnull(FATHER_ID,0) FATHER_ID,CAT_NAME,CAT_LEVEL "+
          " FROM DP_DISPLAY order by ID" ;
        PreparedStatement stmt = con.prepareStatement(sql);
        ResultSet rs = stmt.executeQuery();

        CatNode thisNode = new CatNode(0,"ROOT",null,0,false);
        this.put(0, thisNode );

        while(rs.next()) {
            int code = rs.getInt("ID");
            String name = rs.getString("CAT_NAME");
            int fatherID = rs.getInt("FATHER_ID");
            int level = rs.getInt("CAT_LEVEL");

            boolean isLeaf = true;

            CatNode parentNode = this.get(fatherID);

            thisNode = new CatNode(code,name,parentNode,level,isLeaf);

            if(parentNode.isLeaf)      {
                parentNode.isLeaf = false;
```

```
                    parentNode.children = new ArrayList<CatNode>(5);
                }
                parentNode.children.add(thisNode);
                this.put(code, thisNode );
            }
        }
    }
```

搜索页面用于计数的类：

```
public class CountNode {
    public int no;//分类号
    public String name;//分类名
    public boolean isLeaf;//是否为末级
    public List<CountNode> children = null;
    public CountNode parent;
    public int count;

    public CountNode(int no, String name,CountNode parentNo,boolean isLeaf) {
        this.no = no;
        this.name = name;
        this.parent = parentNo;

        this.isLeaf = isLeaf;
        if (!isLeaf)  {
            children = new ArrayList<CountNode>(5);
        }
    }

    @Override
    public boolean equals(Object o)  {
        if(o instanceof CountNode)  {
                CountNode t = (CountNode)o;
                return (t.no == this.no);
        }
        return false;
    }

    @Override
    public int hashCode(){
        return this.no;
    }
}
```

二级子树展开的实现：

```
//搜索形成的二级子树
HashMap<Integer, CountNode> cat1Set = new HashMap<Integer, CountNode>();
for (Count c : facetCounts) {
    Integer cat2Id = Integer.parseInt(c.getName());
    CatNode cat2Node = catMap.get(cat2Id);
    CountNode newParen = cat1Set.get(cat2Node.parent.no);

    if (limitCat > 0) {
        if (cat2Node.parent.no != limitCat) {
            continue;
        }
    }

    if (newParen == null) {
        newParen = new CountNode(cat2Node.parent.no,
                cat2Node.parent.name, null, false);
        CountNode childNode = new CountNode(cat2Node.no, cat2Node.name,
                newParen, true);
        childNode.count = c.getCount();
        newParen.children.add(childNode);
        cat1Set.put(newParen.no, newParen);
    } else {
        CountNode childNode = new CountNode(cat2Node.no, cat2Node.name,
                newParen, true);
        childNode.count = c.getCount();
        newParen.children.add(childNode);
    }
}
```


使用 MatchAllDocsQuery 返回所有的记录，然后再分类统计，这样可以实现一个分类导航的页面。

有个专门实现分类统计功能的项目 bobo-browse（http://code.google.com/p/bobo-browse/）。不过它需要集成 Spring 框架，用起来麻烦一点。

然后在页面执行搜索时执行如下过程。

```
/**
    * @param cat  当前类别编号
    * @param q  当前查询
    * @return  分类统计列表
    * @throws Exception
  */
public List<CatInf> catCounter(int cat,Query q) throws Exception   {
        CategoryNode thisNode = ListContainer.catMap.get(String.valueOf(cat));
        if(thisNode.isLeaf)   {
            //已经到达最后一级，不能再展开统计
```

```
                return null;
        }

        List<CategoryNode> children = thisNode.children;
        if(children == null) {
            return null;
        }
        ArrayList<CatInf> catList = new ArrayList<CatInf>( children.size() );
        DocSetHitCollector all = new DocSetHitCollector(reader.maxDoc());
        searcher.search(q, all );

        DocSet allDocSet = all.getDocSet();

        String termField = null;//层次列
        if (cat<=0) {
            termField = "hs1";//第 1 层的 ID 号存储在 hs1 列
        } else if (cat<100) {
            termField = "hs2";//第 2 层的 ID 号存储在 hs2 列
        }
        //如果还有后续层，则按前缀匹配来搜
        for (int i=0;i<children.size();++i)     {
                //统计每个子类的搜索结果数
                CategoryNode currentNode = children.get(i);
                DocSetHitCollector these = new DocSetHitCollector(reader.maxDoc());

                searcher.search(
                    new TermQuery(new Term(termField, currentNode.no)),
                    these );
                int count = these.getDocSet().intersectionSize(allDocSet);
                if(count>0) {
                    CatInf catInf = new CatInf();
                    catInf.name = currentNode.name;
                    catInf.no = currentNode.no;
                    catInf.count = count;
                    catList.add(catInf);
                }
        }
        Collections.sort(catList);//对分类统计结果排序后输出
        return catList;
    }
```


可以通过类别编码来控制是否按类别查找，但不推荐这样实现。为了实现 REST 风格的链接，可以直接根据类名按条件查询，如 cat=Music。有些类名可能包含特殊的符号，如“Health & Beauty”。可以使用 URLEncoder 类对类名中的特殊符号转义，例如，把空格转换

成加号，代码如下。

```
URLEncoder.encode(catName,"UTF-8");
```

在 TagLib 中输出在 table 标签中的分类统计结果：

```
public String getCatView() {
    if(_catList == null)
        return "";
    StringBuffer output = new StringBuffer();
    output.append("<table width=\"100%\" border=\"0\" cellspacing=\"0\" cellpadding= \"2\">");
    output.append("<tr> ");
    int count =0 ;
    for(CatInf e : _catList) {
      output.append("<td><a href=\""); //URL 地址
      output.append(_url);
      output.append("?query=");
      output.append(_query);      //查询词
      output.append("&");
      output.append(InitTag.CAT_KEY); //在 URL 中增加一个分类参数用于进一步导航
      output.append("=");
      output.append(e.no);    //类别编号
      output.append("\" class=\"m\">");
      output.append(e.name);    //类别名
      output.append("</a>(");
      output.append(e.count);    //该类别下的文档数量
      output.append(")</td>");
      if(count%3 ==2)   {
          output.append("</tr><tr>");
      }
      count++;
    }
    output.append(" </tr></table>");

    return output.toString();
}
```

最后 JSP 界面通过 Tag 调用 getCatView：

```
<list:prop property="catView"/>
```

6.8.3　相关搜索

一种方法是从搜索日志中挖掘字面相似的词作为相关搜索词列表。首先从一个给定的词语挖掘多个相关搜索词，可以用编辑距离为主的方法查找一个词的字面相似词，如果候选的相关搜索词很多，就要筛选出最相关的 10 个词。下面是利用 Lucene 筛选最相关词的

方法。

```
private static final String TEXT_FIELD = "text";

/**
 *
 * @param words  候选相关词列表
 * @param word  要找相关搜索词的种子词
 * @return
 * @throws IOException
 * @throws ParseException
 */
static String[] filterRelated(HashSet<String> words, String word)       {
      StringBuilder sb = new StringBuilder();

      for(int i=0;i<word.length();++i){
            sb.append(word.charAt(i));
            sb.append(" ");
      }

      RAMDirectory store = new RAMDirectory();
      IndexWriter writer = new IndexWriter(store, new StandardAnalyzer(), true);

      for(String text:words)    {
            Document document = new Document();
            Field textField =
                  new Field(TEXT_FIELD, text, Field.Store.YES, Field.Index.TOKENIZED);
            document.add(textField);
            writer.addDocument(document);
      }
      writer.close();

      IndexSearcher searcher = new IndexSearcher(store);

      QueryParser queryParser = new QueryParser(TEXT_FIELD,
                                                new StandardAnalyzer());
      Query query = queryParser.parse(sb.toString());

      Hits hits = searcher.search(query);
      int maxRet = Math.min(10, hits.length());

      String[] relatedWords = new String[maxRet];
      for (int i = 0; i < maxRet ; i++) {
         Document document = hits.doc(i);
         String text = document.get(TEXT_FIELD);
         System.out.println(text);
```

```
            relatedWords[i]=text;
        }
        searcher.close();
        store.close();

        return relatedWords;
    }
```

整理出这样的相关词表，第一列是关键词，后续是 10 个以内的相关搜索词：

```
集福轩婚礼%集福轩
手机定位跟踪系统%手机定位系统%手机定位%手机定位仪器
喷绘材料卖店电话%我要喷绘材料卖店电话
厦门房产%厦门租房%厦门新闻%厦门桑拿%房产%青岛房产%厦门%恒雄房产
送水果%送水%水果
三星传真机%三星手机
```

另一种方法是，可以把多个用户共同查询的词看成相关搜索词，需要有记录用户 IP 的搜索日志才能实现。类似推荐系统，超市把尿布与啤酒放在一起卖，因为这是关联规则挖掘出的结果。

对这个结果的解释：在美国，一些年轻的父亲下班后经常要到超市去买婴儿尿布，而他们中有 30%～40%的人同时也为自己买一些啤酒。产生这一现象的原因是：美国的太太们常叮嘱她们的丈夫下班后为小孩买尿布，而丈夫们在买尿布后又随手带回了他们喜欢的啤酒。

用户搜索“啤酒”时，提示他是否还要找“尿布”。

ARtool 是一个挖掘关联规则的算法工具集。

然后通过 RelatedEngine 类查找某个关键词的相关词。

```
public static void main(String[] args) throws Exception {
            RelatedEngine re =new RelatedEngine(new File("D:/dic/relatedwords.txt"));
            String word = "徐家汇";
            String[] relatedWords = re.getRelated(word);
            for(String w : relatedWords) {
                System.out.println(w);
            }
}
```

输出相关搜索词如下。

```
上海徐家汇
徐汇
徐家汇价格是
上房徐家汇路附近有吗
```

最后通过自定义的 Tag 标签 RelatedTag 在 JSP 页面显示出相关搜索词。

在标签库描述符中定义 Tag：

```
<tag>
  <name>relatedWords</name>
  <tag-class>com.bitmechanic.listlib.RelatedTag</tag-class>
  <description></description>

  <attribute>
    <name>index</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>

  <attribute>
    <name>url</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>

  <attribute>
    <name>query</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>

</tag>
```


最后在 JSP 页面中引用标签：

```
<list:relatedWords index="D:/search/related" url="Search.jsp" query="<%=query%>"/>
```

6.8.4 再次查找

经常需要从结果中缩小范围再次查找信息。一个实现方法是通过+连接符连接上次查询和当前查询。例如，inputstr 记录了上次查询词，queryString 记录当前查询词，实现代码如下。

```
if (refind) //如果需要再次查找
        queryString = " + ("+queryString+") + ("+inputstr+")";
```

使用这个新的查询词就可以实现再次搜索的功能。

6.8.5 搜索日志

搜索日志是用来分析用户搜索行为和信息需求的重要依据。一般记录如下信息：

- 搜索关键字。
- 用户来源 IP。

- 本次搜索返回结果数量。
- 搜索时间。
- 其他需要记录的应用相关信息。

IP 地址是最容易获取的信息，但其局限性也较为明显：伪 IP、代理、动态 IP、局域网共享同一公网 IP 出口……这些情况都会影响基于 IP 来识别用户的准确性，所以 IP 识别用户的准确性比较低，目前一般不会直接采用 IP 来识别用户。

可以通过 Cookie 记录用户 ID。Cookie 是从用户端存放的 Cookie 文件记录中获取的，这个文件里面一般在包含一个 Cookieid 的同时也会记下用户在该网站的 userid（如果你的网站需要注册登录并且该用户曾经登录过你的网站且 Cookie 未被删除），所以在记录日志文件中 cookie 项时可以优先去查询 Cookie 中是否含有用户 ID 类的信息，如果存在则将用户 ID 写到日志的 Cookie 项，如果不存在则查找是否有 Cookieid，如果有则记录，没有则记为“-”，这样日志中的 Cookie 就可以直接作为最有效的用户唯一标识符被用作统计。当然这里需要注意该方法只有网站本身才能够实现，因为用户 ID 作为用户隐私信息只有该网站才知道其在 Cookie 的设置及存放位置，第三方统计工具一般很难获取。

通过以上的方法实现用户身份的唯一标识后，我们可以通过一些途径来采集用户的基础信息、特征信息及行为信息，然后为每位用户建立起详细的 Profile，具体途径如下。

（1）用户注册时填写的用户注册信息及基本资料。

（2）从网站日志中得到的用户浏览行为数据。

（3）从数据库中获取的用户网站业务应用数据。

（4）基于用户历史数据的推导和预测。

（5）通过直接联系用户或者用户调研的途径获得的用户数据。

（6）有第三方服务机构提供的用户数据。

通过用户身份识别及用户基本信息的采集，我们可以通过网站分析的各种方法在网站实现一些有价值的应用：

- 基于用户特征信息的用户细分；
- 基于用户的个性化页面设置；
- 基于用户行为数据的关联推荐；
- 基于用户兴趣的定向营销。

为了不影响即时搜索的速度，一般不把搜索日志记录直接记录在数据库中，而是写在文本文件中。推荐使用 logback（http://logback.qos.ch/）的日志功能实现。Logback 提供了 3 个 jar 包 Core、Classic、Access。其中 Core 是基础，其他两个包依赖于这个包。logback-classic 是 SLF4J 原生的实现，所以你可以用其他 logging 系统去替换它。当然 logback-classic 依赖于 slf4j-api。logback-access 与 servlet 容器集成。提供 http-access 的 log 功能。SLF4J（http://www.slf4j.org/）几乎已经称为业界日志的统一接口。

这里的项目一共需要 3 个包：slf4j-api-1.6.1.jar、logback-classic-0.9.21.jar 和 logback-core-0.9.21.jar。Logback 通过 logback.xml 进行配置。

这里把当前日志写到 D:/logs/log 文件中，新的一天日志开始时，昨天的日志生成一个新文件。

在搜索类中初始化日志类：

```
private static Logger logger = LoggerFactory.getLogger(SearchBbs.class);
```

当用户执行一次搜索时，记录查询词、返回结果数量、用户 IP 以及查询时间等：

```
logger.info(_query+"|"+desc.count+"|"+"bbs"+"|"+ip);
```

日志文件 log.txt 记录的结果例子如下。

```
什么是新生儿|37|topic|124.1.0.0|2007-11-21 12:25:36
什么是新生儿|28|bbs|124.1.0.0|2007-11-21 12:25:42
怀孕|18|topic|124.1.0.0|2007-11-21 12:26:05
怀孕|2|shangjia|124.1.0.0|2007-11-21 12:26:05
怀孕|145|bbs|124.1.0.0|2007-11-21 12:26:06
怀孕|18|topic|124.1.0.0|2007-11-21 12:30:33
```

这里，第 1 列是用户搜索词，第 2 列是搜索返回结果数量，第 3 列是搜索类别，第 4 列是 IP 地址，第 5 列是搜索的时间。

然后定义搜索日志统计表，如我们需要统计搜索最多的词，可以把搜索最多的词放在 keywordAnalysis 表中：


```
CREATE TABLE [keywordAnalysis] (
        [searchTerms] [varchar] (50)    NOT NULL ,--搜索词
        [AccessCount] [int] NULL ,   --搜索计数
        [Result] [int] NULL       --该词返回结果数
)
```

6.9 查询分析

因为搜索关键词是用户输入的文本信息，所以可以从搜索日志中了解用户使用搜索的意图。有人把 Google 的搜索关键词排行榜称为人类意图数据库。这来源于对搜索日志的分析。

6.9.1 历史搜索词记录

可以通过 Cookie 记录用户经常搜索的关键字，然后就可以从用户经常搜索的关键字来判断用户的兴趣。先看下如何设置用户查询词。Cookie 在用户计算机中以一种类似 map 的方式存放，且只能存放字符串类型的对象。通过 response 对象增加 Cookie，代码如下。

```
Cookie cookie = new Cookie("query", query);
cookie.setMaxAge(60*60*24*30);     //设置 cookie 的存放时间(单位是秒)
//然后通过 response 对象的 addCookie 方法添加 cookie 才能生效
response.addCookie(cookie);
```

通过 request 对象的 getCookies 方法得到一个包含所有 Cookie 的数组。

```
Cookie[] cookies = request.getCookies();
//然后遍历这个数组就能得到记录查询词的 Cookie
```

```
String query = null;
for (int i = 0; i < cookies.length; i++) {
        Cookie c = cookies[i];
        if (c.getName().equals("query")) {
                query = c.getValue();
        }
}
```

上面的例子显示了如何设置并获取名称叫作 query 的 Cookie。如果要记录 5 个查询词，则需要设置 5 个不同的 Cookie。如果要在 Web 界面显示历史查询词，则需要把这些关键词去重，然后再显示出来。

6.9.2　日志信息过滤

公开的搜索会有很多爬虫的访问。搜索日志中包括大量的 Google 爬虫信息，需要把它和普通用户的搜索区分出来。

可以从请求的信息中判断出是哪一种爬虫。例如，baidu 爬虫的“User-Agent”信息：

```
Baiduspider+(+http://help.baidu.jp/system/05.html)
```

Goolge 爬虫的“User-Agent”信息：

```
Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)
```

下面是程序实现：

```
        String userAgent = request.getHeader( "User-Agent" );

        public static String[] getBotName(String userAgent) {
            userAgent = userAgent.toLowerCase();
            int pos=0;
          String res=null;
        if ((pos=userAgent.indexOf("google/"))>-1) {
          res= "Google";
          pos+=7;
    } else
    if ((pos=userAgent.indexOf("msnbot/"))>-1) {
          res= "MSNBot";
          pos+=7;
    } else
    if ((pos=userAgent.indexOf("googlebot/"))>-1) {
          res= "Google";
          pos+=10;
    } else
    if ((pos=userAgent.indexOf("webcrawler/"))>-1) {
          res= "WebCrawler";
          pos+=11;
```

```
        } else
        if ((pos=userAgent.indexOf("inktomi"))>-1) {
            res= "Inktomi";
            pos=-1;
        } else
        if ((pos=userAgent.indexOf("teoma"))>-1) {
            res= "Teoma";
            pos=-1;
        }
        if (res= =null) return null;
        return getArray(res,res,res + getVersionNumber(userAgent,pos));
    }
```

6.9.3 信息统计

可以按搜索次数排列搜索热词，此外还可以按地区来源统计搜索次数。

按地区统计搜索词需要从用户的访问 IP 查询出对应的地址。如表 6-2 所示 IP 地址表记录了一个地区的 IP 范围。

表 6-2　IP 地址表

IP1	IP2	country	city	countryNo	provinceNo
3740518268	3740518268	湖南省永州市	金鹰网吧	1	12
3740518269	3740518400	湖南省永州市	电信	1	12
3740518401	3740518401	湖南省永州市祁阳县	怡心苑网吧	1	12
3740518402	3740518417	湖南省永州市	电信	1	12

下面的代码返回用户 IP 所在省：

```
    String ipin[] = getIp.split("\\.");
    long ipinfo[] = new long[4];
    for (int i = 0; i < ipinfo.length; i++) {
        //从 String 类型的 IP 地址到 long 型的转换可以用查表法更快实现
        ipinfo[i] = Integer.parseInt(ipin[i]);
    }
    long num=ipinfo[0] * 256 * 256 * 256 + ipinfo[1] * 256 * 256 + ipinfo[2] * 256 + ipinfo[3] - 1;
    String sql = "select DIC_Province.caption from ip,DIC_Province where ip.provinceNo = DIC_Province.id
and ip.ip1<="            + num + " and ip.ip2>= " + num + "";
    st = con.createStatement();
    rs = st.executeQuery(sql);
    if (rs.next()) {
        area = rs.getString(1);
    }
```

首先建立搜索日志统计表：

```
CREATE TABLE SC_SEARCH_STAT (
        ID NUMERIC(12 , 0)    IDENTITY ,                    //自增长 ID
        SEARCH_WORD VARCHAR(90)    NULL,                    //搜索词
        SEARCH_NUM INT    NULL,                             //搜索次数
        SEARCH_DATE DATE    NULL,                           //搜索日期
        SEARCH_RESULT INT    NULL                           //搜索词的返回结果数量
    )
```

搜索统计表每天更新一次。每次把上一天的搜索日志文件中的数据统计后写入该表。搜索次数 SEARCH_NUM 指一天内搜索 SEARCH_WORD 这个词的独立 IP 的统计数量，同日、同地址、同搜索词只算一次。搜索统计用到的主要数据结构如下。

```
HashMap<String,HashSet<String>> word2IP =
                    new HashMap<String,HashSet<String>>();//搜索词到 IP 的映射
HashMap<String,Integer> word2ResultNum =
                    new HashMap<String,Integer>();//搜索词到搜索结果数的映射
…
//统计信息
HashSet<String> ips = word2IP.get(key);//如果搜索词已存在
if (ips!=null) {
                ips.add(strIP);//增加当前 IP
                word2ResultNum.put(key, resultNum);
} else if (resultNum > 0) {//如果搜索返回结果数大于零
                ips = new HashSet<String>();
                ips.add(strIP);
                word2IP.put(key, ips);
                word2ResultNum.put(key, resultNum);
}
…
//统计信息写入统计表
String sql =
"insert into SC_SEARCH_STAT(SEARCH_WORD,SEARCH_NUM,SEARCH_DATE, SEARCH_RESULT)
values(?,?,?,?)";
PreparedStatement pstmt = con.prepareStatement(sql);

for (Entry<String, HashSet<String>> e : word2IP.entrySet()) {
        pstmt.setString(1, e.getKey());
        pstmt.setInt(2, e.getValue().size());
        pstmt.setString(3, yesDate);
        pstmt.setInt(4, word2ResultNum.get(e.getKey()));
        pstmt.executeUpdate();
}
```

6.9.4 挖掘日志信息

可以从不同的角度挖掘搜索日志。

- 挖掘单个词。可以挖掘出用户对查询语法的使用情况，例如，filetype、site 等查询语法。可以统计用户搜索中使用空格分开多个词来搜索的比例。
- 或者按用户会话（Session）挖掘词。有的用户先搜索“工程电磁学基础（第 6 版）”，没有结果返回，几秒钟后他换了一个搜索词“工程电磁学基础”，这次有 22 条结果返回。因此可以把用户对查询词的修改分为四种情况：减少查询词、增加查询词、部分替换查询词、完全更换查询词。另外，还可以统计用户搜索词之间的词序，考虑上一个词到下一个词之间的转移概率。
- 根据用户与词之间的关联可以挖掘出词与词之间的关联或者用户与用户之间的关联。观察到一个用户搜索“眉笔”后，又搜索了“粉底液”和“紧肤水”，这些词都是美容相关的商品名，所以考虑使用关联规则挖掘出相关词。可以根据用户的搜索词对用户分类，计算出用户对每个类别的隶属度。

6.9.5 查询词意图分析

用户输入 138777，也许是想找以这个数字串开始的手机号码。用户输入“张”，也许是想找所有姓“张”的人。

6.10 部署网站

服务器端操作系统推荐采用 Linux 操作系统。Linux 有各种版本，这里以免费的 CentOS 为例。用静态 IP 地址配置一个网络连接的 IPv4 属性。例如，静态 IP 地址是 201.147.214.149。eth0 配置文件的内容如下。

```
# cat /etc/sysconfig/network-scripts/ifcfg-eth0
TYPE=Ethernet
DEVICE=eth0
HWADDR=00:2g:fc:1b:c3:9e
ONBOOT=yes
USERCTL=no
IPV6INIT=no
PEERDNS=yes
NETMASK=255.255.255.128
IPADDR=201.147.214.149
GATEWAY=201.147.214.254
```

重新启动网络服务：

```
#service network restart
```

如果托管的机器要换机柜。可以远程先修改好网卡的配置，然后让机房人员挪动机器，重新启动机器。同时修改 DNS 中的 IP 地址。

6.10.1　部署到 Web 服务器

配置 Java 环境，设置环境变量 JAVA_HOME 和 PATH 的值。修改脚本文件/etc/bashrc：

```
#vi /etc/bashrc。
```

增加如下行：

```
export JAVA_HOME=/usr/local/jdk1.6.0_21
export PATH=$JAVA_HOME/bin:$PATH
```

从 Tomcat 官方网站 http://tomcat.apache.org/下载 tar.gz 包。

```
# wget
http://www.fayea.com/apache-mirror/tomcat/tomcat-7/v7.0.33/bin/apache-tomcat-7.0.33.tar.gz
```

解压缩这个文件：

```
#tar -xf apache-tomcat-7.0.33.tar.gz
```

然后增加 Tomcat 所使用的内存。修改配置文件 catalina.sh：

```
#vi /usr/local/apache-tomcat-6.0.32/bin/catalina.sh
```

在文件 catalina.sh 的开始位置增加如下行：

```
JAVA_OPTS=-Xmx1024m
```

修改 Tomcat 配置文件 server.xml，把监听端口号从 8080 改到 80，并且支持 UTF-8 编码：

```
#vi /usr/local/apache-tomcat-6.0.32/conf/server.xml
```

增加配置：

```
useBodyEncodingForURI="true" URIEncoding="UTF-8"
```

可以把 Web 应用打一个 war 包，然后传到服务器上的 webapps/子路径下，会自动解压缩 war 包中的 Web 应用。

也可以压缩开发环境中的文件：

```
#tar -cjf price.tar.bz2 ./price
```

在正式环境中下载压缩好的文件 price.tar.bz2，然后解压缩文件：

```
#tar -xjf price.tar.bz2
```

启动 Tomcat：

```
#startup.sh
```

查看 Tomcat 是否已经运行了：

```
#pgrep -l java
```

或者使用命令：

```
#ps -ef |grep java
```

虚拟主机服务器提供商可能提供这样的服务：当 Tomcat 服务停止时，会自动运行一个 Apache 显示错误信息。启动 Tomcat 之前，先停止 Apache 服务：

```
#httpd -k stop
```

查看 Apache 服务是否已经停止了：

```
#pgrep httpd
```

如果需要更好的性能，可以使用 Resin。处理静态页面，Resin 比 ngnix 或 Apache 快。http://www.caucho.com/download/下载 Resin 免费版本。

在 bin 目录下面，用 vi 命令新建一个名为 startResin.sh 的文件：


```
vi ./startResin.sh
```

在文件内输入如下信息：

```
export LC_ALL=zh_CN.GB18030
export LANG=zh_CN.GB18030
nohup ./httpd.sh & -Xms512M -Xmx1024M
```

备份到目录/home/webbak/ROOT2012：

```
cp -r ./ROOT/    /home/webbak/ROOT2012
```

启动 Resin 4。

```
resin.sh start
```

如果需要，可以给网站买一个好记的域名。从域名供应商购买域名后，增加 DNS 信息。如果修改 DNS 信息后，要清空本地的 DNS 缓存信息。Windows 下使用如下的命令清空缓存：

```
>ipconfig /flushdns
```

查询一个域名的 A 记录：

```
>nslookup www.lietu.com 198.153.192.1
```

6.10.2 防止攻击

当站点无法访问了，可能是被攻击了。一种常见的攻击称为分布式拒绝服务攻击，英文是 Distributed Denial Of Service，简称 DDOS。检查服务器是否在 DDOS 状态的一个快速

而有用的命令是：

```
#netstat -anp |grep 'tcp\|udp' | awk '{print $5}' | cut -d: -f1 | sort | uniq -c | sort -n
```

这会列出花费最多的连接到服务器的 IP。但要记住，DDOS 正在变得更复杂，它可能用更多的 IP 地址，每个 IP 地址使用更少的连接。例如，使用代理 IP。如果是这样，即使在 DDOS 下，仍然只有很少数量的连接。这称为 CC 攻击。

另外一个非常重要的事情是看有多少正在处理的活跃的连接。

```
#netstat -n | grep :80 |wc -l
```

这将显示活跃的连接数。许多攻击通常开始一个到服务器的连接，然后不发送任何答复，让服务器等待到超时。活动连接的数量会有很大的不同，但如果是 500 以上，就可能有问题。

```
#netstat -n | grep :80 | grep SYN |wc -l
```

如果超过 100，就有 SYN 攻击的麻烦。大量的 SYN 请求导致未连接队列被塞满，使正常的 TCP 连接无法顺利完成三次握手，通过增大未连接队列空间可以缓解这种压力。

Linux 用变量 tcp_max_syn_backlog 定义 backlog 队列容纳的最大半连接数。在 Redhat AS 中，这个值默认是 1024。这个值是远远不够的，一次强度不大的 SYN 攻击就能使半连接队列占满。可以通过以下命令修改此变量的值。


```
# sysctl -w net.ipv4.tcp_max_syn_backlog="2048"
```

在 Linux 下可以通过修改/etc/sysctl.conf，添加下列选项达到效果。

```
# add by geminis for syn crack
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_max_syn_backlog=2048
net.ipv4.tcp_synack_retries=1
```

Linux 有一个很好的工具来拒绝为“不怀好意”的 IP 提供服务，称为 iptables。

很多管理员害怕使用 iptables。阻塞 IP 会阻塞这个 IP 访问任何服务器资源，不仅是 Web 服务器，还包括 FTP、telnet 等。不幸的是，很多系统管理员和网站站长害怕使用 iptables，因此没有经验。如果你直接编辑 iptables 的配置文件，可能会导致宕机。例如，一个语法错误，可能会阻止你访问 SSH、FTP、HTTP 和任何其他的服务。因此，不要直接编辑配置文件 iptables-config。最好从 Linux 命令行进入 iptables 命令。如果有语法错误，命令行接口会拒绝这个命令。下面是一些常用的例子。

阻塞从 120.60.0.0 到 120.60.255.255 范围内的 IP。

```
#iptables -I INPUT -m iprange --src-range 120.60.0.0-120.60.255.255 -j DROP
```

要阻止一个 IP，如 120.60.43.201，使用下面的命令：

```
#iptables -A INPUT -s 120.60.43.201 -j DROP
```

显示当前的 iptables 文件，而不编辑它，使用下面的命令：

```
#iptables -L
```

iptables 不能自动屏蔽恶意 IP，只能手动屏蔽。一个轻量级的脚本 DDOS deflate 能够自动屏蔽 DDOS 攻击者的 IP。

可以配置白名单的 IP 地址，通过配置：/usr/local/ddos/ignore.ip.list。

IP 地址被封时间是预先设定的，默认 600 秒后自动解除封锁。通过配置文件，脚本可以定时周期性运行（默认是 1 分钟）。有 IP 地址被封锁时，可以为指定的邮箱接收电子邮件警报。

这些都可以写在配置文件/usr/local/ddos/ddos.conf 中。

安装 DDOS deflate：

```
# wget http://www.inetbase.com/scripts/ddos/install.sh
# chmod 0700 install.sh
# ./install.sh
```

下面解释一下 DDOS deflate 脚本主要配置文件 ddos.conf：

```
##### Paths of the script and other files
PROGDIR="/usr/local/ddos"              //文件存放目录
PROG="/usr/local/ddos/ddos.sh"         //主要功能脚本
IGNORE_IP_LIST="/usr/local/ddos/ignore.ip.list"          //白名单地址列表
CRON="/etc/cron.d/ddos.cron"           //crond 定时任务脚本
APF="/etc/apf/apf"
IPT="/sbin/iptables"

##### frequency in minutes for running the script
##### Caution: Every time this setting is changed, run the script with --cron
#####              option so that the new frequency takes effect
FREQ=1          //间隔多久检查一次，默认为 1 分钟

##### How many connections define a bad IP? Indicate that below.
NO_OF_CONNECTIONS=150      //最大连接数设置，超过这个数字的 IP 就会被屏蔽，默认即可

##### APF_BAN=1 (Make sure your APF version is atleast 0.96)
##### APF_BAN=0 (Uses iptables for banning ips instead of APF)
APF_BAN=0          //1：使用 APF，0：使用 iptables，推荐使用 iptables

##### KILL=0 (Bad IPs are'nt banned, good for interactive execution of script)
##### KILL=1 (Recommended setting)
KILL=1       //是否屏蔽 IP，默认即可

##### An email is sent to the following address when an IP is banned.
##### Blank would suppress sending of mails
EMAIL_TO="root"          //发送电子邮件报警的邮箱地址，换成自己使用的邮箱即可
```

```
##### Number of seconds the banned ip should remain in blacklist.
BAN_PERIOD=600          //屏蔽 IP 的时间，根据情况调整
```

最后开启系统 crond 服务即可。

执行 uninstall.ddos 来卸载脚本：

```
# wget http://www.inetbase.com/scripts/ddos/uninstall.ddos
# chmod 0700 uninstall.ddos
# ./uninstall.ddos
```

更好的方法是安装 CSF（Config Security Firewall），它可以在极大程度上保护服务器安全。CSF 是可以免费使用的，基于 iptables 的防火墙，很容易集成到 CPanel。CPanel 是为网站所有者设计的一套 Web 形式的控制系统，网站所有者甚至可以直接把它当作网站后台 Windows 操作系统。

银行为了防止黑客暴力破解持卡人的密码，采用连续三次输入密码错误就锁定该账户的方法。CSF 防火墙为了防止暴力破解密码，也会自动屏蔽连续登录失败的 IP。

可以通过它管理网络端口，只开放必要的端口。可以免疫小流量的 DDOS 和 CC 攻击。

CentOS 下需要先安装 CSF 依赖包：

```
yum install perl-libwww-perl perl iptables
wget http://www.configserver.com/free/csf.tgz

tar zxf csf.tar.gz
```

如果和 Apache 服务器配合使用，则执行：

```
sh ./csf/install.sh
```

如果直接管理防火墙，则执行：

```
sh ./csf/install.directadmin.sh
```

按照安装程序的指示装好后，可以运行测试程序：

```
perl /etc/csf/csftest.pl
```

如果没问题就可以启动防火墙：

```
csf -s
```

重新启动防火墙：

```
csf -r
```

刷新规则，或者停止防火墙：

```
csf -f
```

6.11 本章小结

本章介绍了使用 Java 框架 Spring 实现的搜索界面，并且介绍了很多重要而且基本的搜索功能界面实现，如复杂条件搜索界面和用户输入提示词、分类查找界面等。

HTTP PATCH 方法在 2010 年 3 月才成为正式的方法，Spring 3.2 开始支持 PATCH 方法。

因为不需要重复输入，所以在浏览器的输入框中提交查询词到一个搜索引擎，需要花费的时间更少一些，但仍然需要搜索结果有一定用处。

因为有的浏览器禁用 JavaScript，所以尽量用普通的 HTML 标签，只是在必要时才用 JavaScript。

2010 年前使用 Script.aculo.us 实现自动完成功能，当 jQuery 开始流行后，依赖 Prototype 的 Script.aculo.us 不再流行。等 WebAssembly 技术逐渐成熟后，前端界面开发会更加简单。

除了 Spring，还可以使用 Struts2 来实现搜索界面。Struts2 并没有退出历史舞台，它的版本仍然在更新中。另外，还可以使用 Spark（http://sparkjava.com/）网站开发框架。

在界面设计上，要想办法节约用户的时间。例如，手机上的锁状态，需要用户额外输入才能解锁，用触感指纹或者手的设计代替。

有些热词直接跳转，如搜索手机直接跳转到相关手机的界面，而不执行相关性查询。有个散列表对应查询词和跳转的页面。

新闻搜索结果页中显示小的缩略图片。因为 HTML5 中的 src 属性可以直接保存表示图片的二进制序列，所以把这个二进制序列直接放入索引中。这样使用<img>标签：

```
<img src="...">
```

这里，xxxxx...部分是 Gif 图片数据的 base64 编码。

看搜索日志可能会想，搜索访问量比卖快餐的量少太多了。用户难得搜索一次，所以要把返回页面的价值最大化。返回结果的信息可能是各种各样的，但一定都是用户可能想看到的信息，尽量不要列出任何无关的信息。现在各大搜索引擎的第一页基本上都是综合页，同时返回新闻、图片或视频的搜索结果都是聚合搜索（Aggregated Search）了。还有类似于 Naver（http://www.naver.com）这样的综合页面。

在搜索结果页显示个性化的背景图。背景图根据什么规则来替换呢？比如说喜欢足球的用足球背景小图标；喜欢户外活动的用风景图小图标；喜欢书法的用毛笔；女性用鲜花……类似博客背景那样的。

如何判断他喜欢足球呢？根据不同 profile 推出不同查询结果页背景，类似于根据用户特征数据做个性化推荐。

OCR 文字识别

对图片中的文字，可以使用 OCR（Optical Character Recognition，光学字符识别）软件包识别出来，然后再建立索引。

7.1 Tesseract

Leptonica 是一个开源库，其中包含了对图像处理和图像分析应用程序的软件。因为 Tesseract 依赖 Leptonica，所以如果要在命令行运行 Tesseract 可以先安装 Leptonica，然后再安装 Tesseract。

使用 JavaCPP Presets 提供的 Tesseract Java API。首先从 https://github.com/bytedeco/javacpp-presets 下载二进制版本的 JavaCPP Presets，然后根据操作系统类型在项目中引入 5 个包，如在 Windows 下可以引入：javacpp-1.3.jar、leptonica-1.73-1.3.jar、leptonica-windows-x86.jar、tesseract-3.04.01-1.3.jar、tesseract-windows-x86.jar。

有各种语言预训练好的模型文件。如果识别英文，则需要有英文的模型文件。识别英文字符的代码如下。

```
import org.bytedeco.javacpp.BytePointer;
import org.bytedeco.javacpp.lept.PIX;
import org.bytedeco.javacpp.tesseract.TessBaseAPI;
import static org.bytedeco.javacpp.lept.pixRead;
import static org.bytedeco.javacpp.lept.pixDestroy;

public class MinimalExample {

    public static void main(String[] args) {
        TessBaseAPI api = new TessBaseAPI();
        if (api.Init(".", "ENG") != 0) { //加载识别英文字符的模型文件
            System.err.println("Could not initialize tesseract.");
            System.exit(1);
        }

        String filePath = "03-27-15-33-15.jpg";
```

```
            //用 leptonica 库打开输入图像
            PIX image = pixRead(filePath);
            api.SetImage(image);
            //获取 OCR 结果
            BytePointer outText = api.GetUTF8Text();
            String string = outText.getString();
            System.out.println("OCR output:\n" + string);

            //销毁用过的对象并释放内存
            api.End();
            outText.deallocate();
            pixDestroy(image);
        }
    }
```

可以调用 ResultIterator.BoundingBox(RIL_WORD, coord1, coord2, coord3, coord4)方法按块显示识别的内容。这里的 RIL_WORD 可以调整成按单词、句子和段落进行迭代的迭代器级别。按单词迭代的完整代码如下。


```
BytePointer outText;

TessBaseAPI api = new TessBaseAPI();
if (api.Init(".", "eng") != 0) {
    System.err.println("Could not initialize tesseract.");
    System.exit(1);
}

PIX image = pixRead("03-27-15-33-17.jpg");
api.SetImage(image);
//得到 OCR 结果
outText = api.GetUTF8Text();
System.out.println("OCR output:\n" + outText.getString());

final ResultIterator ri = api.GetIterator();

int x1 = 0;
int y1 = 0;
int x2 = 0;
int y2 = 0;

IntPointer coord1 = new IntPointer(x1);
IntPointer coord2 = new IntPointer(y1);
IntPointer coord3 = new IntPointer(x2);
IntPointer coord4 = new IntPointer(y2);

ri.Begin();
```

```
if (ri != null) {

    do {
        BytePointer word = ri.GetUTF8Text(tesseract.RIL_WORD);
        float conf = ri.Confidence(tesseract.RIL_WORD);
        boolean box = ri.BoundingBox(tesseract.RIL_WORD, coord1,
                coord2, coord3, coord4);

        System.out.println(word.getString());
        System.out.println(coord1.get()+" "+coord2.get()+" "+coord3.get()+" "+coord4.get());
        System.out.println(conf);
        System.out.println(box);

    } while (ri.Next(tesseract.RIL_WORD));

}

api.End();
outText.deallocate();
pixDestroy(image);
```


限定识别的字符范围：

```
TessBaseAPI tesseract = new TessBaseAPI();
String dataPath = ".";
String language = "eng";
int initRet = tesseract.Init(dataPath, language);
tesseract.SetVariable("tessedit_char_whitelist",
        "BCDFGHJKLMNPQRSTVWXYZ0123456789-");
tesseract.SetVariable("language_model_penalty_non_freq_dict_word", "1");
tesseract.SetVariable("language_model_penalty_non_dict_word ", "1");
tesseract.SetVariable("load_system_dawg", "0");
```

下载模型文件：

```
# git clone https://github.com/tesseract-ocr/tessdata.git
```

使用识别中文的文件 chi_sim.traineddata：

```
TessBaseAPI api = new TessBaseAPI();
int ret = api.Init(".", "chi_sim");
```

多线程执行 OCR 任务：

```
public class OCRTask implements Callable<String> {
    String imgFile;

    public OCRTask(String f) {
```

```
            imgFile = f;
        }

        @Override
        public String call() throws Exception {
            TessBaseAPI api = new TessBaseAPI();
            if (api.Init(".", "eng") != 0) {
                System.err.println("Could not initialize tesseract.");
                return null;
            }
            PIX image = pixRead(imgFile);
            api.SetImage(image);
            BytePointer outText = api.GetUTF8Text();
            String string = outText.getString();
            outText.deallocate();
            pixDestroy(image);
            api.End();
            return string;
        }
    }
```


除了 JavaCPP Presets，也可以使用 Tess4J(http://tess4j.sourceforge.net/)。调用 Tess4J 的代码如下。

```
File imageFile = new File("eurotext.tif");
ITesseract instance = new Tesseract();

String result = instance.doOCR(imageFile);
System.out.println(result);
```

OpenCV 3 整合 TesseractOCR 的 API 不在基本包中，而是放在 opencv_contrib 项目的 text 模块里。目前只有 C++和 Python 的实现代码。

如果希望提高准确度，可以自己训练模型。生成训练图片的 Java 代码如下。

```
public static void createImage(String str, Font font, File outFile,
            Size size) throws Exception {
        int width = size.width;
        int height = size.height;

        //创建图片
        BufferedImage image = new BufferedImage(width, height,
                BufferedImage.TYPE_INT_BGR);
        Graphics g = image.getGraphics();
        g.setClip(0, 0, width, height);
        g.setColor(Color.white);
        g.fillRect(0, 0, width, height);//先用白色填充整张图片,也就是背景
        g.setColor(Color.black);        //再换成黑色
```

```
        g.setFont(font);                                    //设置画笔字体
        FontMetrics fm = g.getFontMetrics(font);
        int stringWidth = fm.stringWidth(str);
        int x = (width - stringWidth) / 2;
        int ascent = fm.getAscent();
        int descent = fm.getDescent();
        /** 用于获得垂直居中的 y */
        Rectangle clip = g.getClipBounds();
        int y = (clip.height - (ascent + descent)) / 2 + ascent;
        g.drawString(str, x, y);                            //画出字符串
        g.dispose();
        ImageIO.write(image, "png", outFile);    //输出 png 图片
    }
```

7.2 使用 TensorFlow 识别文字

项目首先导入 jar 包 tensorflow-1.2.1.jar、libtensorflow-1.2.1.jar 和 libtensorflow_jni-1.2.1.jar。然后测试：


```
//输出使用的 TensorFlow 版本号
System.out.println("I'm using TensorFlow version: " +    TensorFlow.version());
```

TensorFlow 软件使用 Tensor 来表示计算中的数据。Tensor 类是一个带类型的多维数组。例如，一个 64 位的整数标量：

```
Tensor s = Tensor.create(62L);
```

一个一维向量：

```
float[] x = new float[]{1,2,3,4,5,6,7,8,9,0,1,2,30};
Tensor tx = Tensor.create(x);
System.out.println(tx);   //输出 Tensor 的形状
```

一个 3×2 的浮点数矩阵：

```
float[][] matrix = new float[3][2];
Tensor m = Tensor.create(matrix);
System.out.println(m.numDimensions()); //输出维度
```

执行加法的例子：

```
Graph g = new Graph();
Session s = new Session(g);

Tensor node1 = Tensor.create(1.0f);
Tensor node2 = Tensor.create(4.0f);

System.out.println(node1.floatValue());
```

```
System.out.println(node2.floatValue());
//两个常量
Operation o1 =
      g.opBuilder("Const", "node1").setAttr("dtype", node1.dataType())
       .setAttr("value", node1).build();
Operation o2 =
      g.opBuilder("Const", "node2").setAttr("dtype", node2.dataType())
         .setAttr("value", node2).build();
Operation node3 =
      g.opBuilder("Add", "sum").addInput(o1.output(0)).addInput(o2.output(0)).build();
System.out.println(node3);
List<Tensor> results = s.runner().fetch("sum").run();
System.out.println(results.get(0).floatValue());
s.close();
```

TensorFlow Java API 最适合导入和执行模型。它确实支持构建 TensorFlow 操作的图形，但没有像 Python 那样有很多更高级别的便利 API（图层库、估计器类、优化器等）。因此，目前来说，建议在 Python 中构建图形，然后将它们导入到 Java 应用程序中。

使用已经训练好的模型文件执行分类的代码如下。

```
//加载模型文件
String modelpath = "d:/test/";
Path path = Paths.get(modelpath, "tensorflow_inception_graph.pb");
byte[] graphDef = Files.readAllBytes(path);

//对图片分类
String imagepath = "data/lion.png"; //输入的图片名称
byte[] imageBytes = Files.readAllBytes(Paths.get(imagepath));
Tensor image = Tensor.create(imageBytes);
Graph g = new Graph();
g.importGraphDef(graphDef);
Session s = new Session(g);
Tensor result =
          s.runner().feed("DecodeJpeg/contents", image).fetch("softmax").run().get(0);
final long[] rshape = result.shape();
int nlabels = (int) rshape[1];
float[] probabilities = result.copyTo(new float[1][nlabels])[0];
System.out.println("probabilities len: " + probabilities.length);
System.out.println("label: " + maxIndex(probabilities));
s.close();
```

使用 TensorFlow 预测图像类别完整的代码如下。

```
public static void main(String[] args) {
      if (args.length != 2) {
         printUsage(System.err);
         System.exit(1);
```

```
        }
        String modelDir = args[0];   //模型文件的路径
        String imageFile = args[1];   //图像文件

         //读入训练好的模型文件
        byte[] graphDef =
           readAllBytesOrExit(Paths.get(modelDir, "tensorflow_inception_graph.pb"));
        List<String> labels =
             readAllLinesOrExit(Paths.get(modelDir,
                 "imagenet_comp_graph_label_strings.txt"));                    //读入标注字符串
        byte[] imageBytes = readAllBytesOrExit(Paths.get(imageFile));         //读入图像文件

        try (Tensor image = constructAndExecuteGraphToNormalizeImage(imageBytes)) {
           float[] labelProbabilities = executeInceptionGraph(graphDef, image);
           int bestLabelIdx = maxIndex(labelProbabilities);
           System.out.println(
                 String.format(
                      "BEST MATCH: %s (%.2f%% likely)",
                      labels.get(bestLabelIdx), labelProbabilities[bestLabelIdx] * 100f));
        }
    }
```


Java 的 importGraphDef()函数只导入计算图（在 Python 代码中由 tf.train.write_graph 写出），它没有加载经过训练的变量（存储在检查点中）的值。

TensorFlow 的 SavedModel 格式包括有关模型（图形、检查点状态、其他元数据）的所有信息。因此，如果想在 Java 中使用，可以使用 SavedModelBundle.load 创建使用训练变量值初始化的会话。

Python 中的保存模型代码如下。

```
    def save_model(session, input_tensor, output_tensor):
       signature = tf.saved_model.signature_def_utils.build_signature_def(
          inputs = {'input': tf.saved_model.utils.build_tensor_info(input_tensor)},
          outputs = {'output': tf.saved_model.utils.build_tensor_info(output_tensor)},
       )
       b = saved_model_builder.SavedModelBuilder('/tmp/model')
       b.add_meta_graph_and_variables(session,
                                      [tf.saved_model.tag_constants.SERVING],
       signature_def_map={tf.saved_model.signature_constants.    DEFAULT_SERVING_SIGNATURE_DEF_KEY:
signature})
       b.save()
```

然后在 Java 中加载模型：

```
    try (SavedModelBundle b = SavedModelBundle.load("/tmp/mymodel", "serve")) {
       // b.session().run(...)
    }
```

7.3 OpenCV

对于证件中的文字识别，可以先识别人脸，然后用锐化、二值化和平滑等各种滤波器处理图像，最后用最小外接矩形框出包含文字的区域。OpenCV 中包含了现成的人脸识别功能。

介绍在 Windows 下安装 OpenCV 3.x。首先从 OpenCV 官方网站（http://opencv.org）下载 OpenCV 库（3.x 版）opencv-3.x.0-vc14.exe，然后在选择的位置提取下载的 OpenCV 文件。为了用 Java 调用 OpenCV，只需要两个文件：位于\opencv\build\java 的 opencv-3xx.jar 文件和位于\opencv\build\java\x64（用于 64 位系统）或 \opencv\build\java\x86（对于 32 位系统）的 opencv_java3xx.dll 库。 每个文件的 3xx 后缀是当前 OpenCV 版本的快捷方式，例如，OpenCV 3.0 是 300，而 OpenCV 3.2 是 320。在 Eclipse 中把这两个文件加到用户库中，其中的 dll 加到本地库路径。

Mat 是一个多维的密集数据数组，可以用来处理向量和矩阵、图像、直方图等常见的多维数据。创建一个 5×5 的单位矩阵代码如下。


```
//首先记得加载 dll
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
System.out.println("OpenCV Version:" + Core.VERSION);
Mat m = Mat.eye(5, 5, CvType.CV_8UC1);

System.out.println("mat = " + m.dump());
```

因为 OpenCV 不支持读入所有的图片格式，所以首先验证是否正确读入：

```
String inputImg = "shucai-1.jpg";
Mat image = Imgcodecs.imread(inputImg);
System.out.println("empty?"+image.empty()); //如果 image 是空的，则读入错误
```

如果读入错误，则可以利用 Java 本身的图像读入功能。为了能够利用 Java 的图像处理功能，需要实现 Mat 和 BufferedImage 互相转换。为了把 BufferedImage 转换成 Mat，首先创建一个新的 Mat：

```
Mat newMat = Mat(rows, cols, type);
```

然后从 BufferedImage 中获取像素值，并把它放入 newMat：

```
newMat.put(row, col, pixel);
```

完整的代码如下。

```
public static Mat img2Mat(BufferedImage in) {
    Mat out;
    byte[] data;
    int r, g, b;
    int height = in.getHeight();
    int width = in.getWidth();
```

```
        if (in.getType() == BufferedImage.TYPE_INT_RGB
                || in.getType()== BufferedImage.TYPE_INT_ARGB) {
            out = new Mat(height, width, CvType.CV_8UC3);
            data = new byte[height * width * (int) out.elemSize()];
            int[] dataBuff = in.getRGB(0, 0, width, height, null, 0, width);
            for (int i = 0; i < dataBuff.length; i++) {
                data[i * 3 + 2] = (byte) ((dataBuff[i] >> 16) & 0xFF);
                data[i * 3 + 1] = (byte) ((dataBuff[i] >> 8) & 0xFF);
                data[i * 3] = (byte) ((dataBuff[i] >> 0) & 0xFF);
            }
        } else {
            out = new Mat(height, width, CvType.CV_8UC1);
            data = new byte[height * width * (int) out.elemSize()];
            int[] dataBuff = in.getRGB(0, 0, width, height, null, 0, width);
            for (int i = 0; i < dataBuff.length; i++) {
                r = (byte) ((dataBuff[i] >> 16) & 0xFF);
                g = (byte) ((dataBuff[i] >> 8) & 0xFF);
                b = (byte) ((dataBuff[i] >> 0) & 0xFF);
                data[i] = (byte) ((0.21 * r) + (0.71 * g) + (0.07 * b)); // luminosity
            }
        }
        out.put(0, 0, data);
        return out;
    }
```


将 Mat 转换为 BufferedImage：

```
    public static BufferedImage mat2Img(Mat mat) {
        byte[] data = new byte[mat.width() * mat.height()
                * (int) mat.elemSize()];
        int type;
        mat.get(0, 0, data);

        switch (mat.channels()) {
        case 1:
            type = BufferedImage.TYPE_BYTE_GRAY;
            break;
        case 3:
            type = BufferedImage.TYPE_3BYTE_BGR;
            // bgr to rgb
            byte b;
            for (int i = 0; i < data.length; i = i + 3) {
                b = data[i];
                data[i] = data[i + 2];
                data[i + 2] = b;
            }
            break;
```

```
        default:
            throw new IllegalStateException("Unsupported number of channels");
        }

        BufferedImage out = new BufferedImage(mat.width(), mat.height(), type);
        out.getRaster().setDataElements(0, 0, mat.width(), mat.height(), data);

        return out;
    }
```

7.3.1 预处理

对于大图片，可以对输入图像向下采样：

```
Mat image = Imgcodecs.imread("info1.jpg");
Mat rgb = new Mat();

Imgproc.pyrDown(image, rgb);

String outputfile1 = "t1.jpg";
Imgcodecs.imwrite(outputfile1, rgb); //图像写入到文件
```


或者将图像缩放到固定尺寸：

```
Mat image = Imgcodecs.imread("info1.jpg");
Mat rgb = new Mat();
Imgproc.resize(image, rgb ,new Size(1920, 1080));
String outputfile1 = "t2.jpg";
Imgcodecs.imwrite(outputfile1, rgb);
```

Imgproc.resize 方法原型如下。

```
public static void resize(Mat src,
                          Mat dst,
                          Size dsize,
                          double fx,
                          double fy,
                          int interpolation)
```

使用 dsize 或缩放因子，不要两者都用。
下面的代码把图像缩放到不超过 1000×1000 像素大小。

```
Mat image = Imgcodecs.imread("info1.jpg");
float ra = (float) 1000 / Math.max(image.width(), image.height());
Size dsize = new Size(0, 0);
Mat output = new Mat();
Imgproc.resize(image, output, dsize, ra, ra, Imgproc.INTER_LINEAR);
```

可以通过锐化让包含文字的区域变得更容易识别：

```
//变形核
Mat morphKernel1 = Imgproc.getStructuringElement(Imgproc.MORPH_RECT, // 矩形
        new Size(4, 1.8));

//用来存锐化后的图像对象
Mat dilatepic = new Mat();
Imgproc.dilate(bw, dilatepic, morphKernel1);  //实现锐化
String outputfile3 = "deskew/dilatepic.jpg";
Imgcodecs.imwrite(outputfile3, dilatepic);
```

转换成灰度图：

```
Mat grad = new Mat();
Imgproc.cvtColor(image, grad, Imgproc.COLOR_BGR2GRAY);
```

可以使用固定阈值实现二值化：

```
Imgproc.threshold(grad, bw, 102.0, 255.0, Imgproc.THRESH_BINARY); //阈值是 102
```


也可以使用 OTSU 算法自动实现二值化：

```
Imgproc.threshold(src_gray, bw, 0.0, 255.0, Imgproc.THRESH_BINARY
        | Imgproc.THRESH_OTSU);
```

先二值化，然后使用变形核执行腐蚀操作。Imgproc.getStructuringElement 方法可以获取矩形、椭圆或十字形的结构元素（变形核），代码如下。

```
Mat bw = new Mat(); //用来存二值化的图

Imgproc.threshold(grad, bw, 0.0, 255.0, Imgproc.THRESH_BINARY
        | Imgproc.THRESH_OTSU);

//变形核
Mat morphKernel = Imgproc.getStructuringElement(Imgproc.MORPH_RECT, //矩形
        new Size(7, 10)); //宽度 7,高度 10

//用来存腐蚀后的图像对象
Mat erodedpic = new Mat();
//进行腐蚀
Imgproc.erode(bw, erodedpic, morphKernel);
Imgcodecs.imwrite("erode.jpg", erodedpic);
```

为了在一幅图像里得到轮廓区域的参数，可以调用 Imgproc.findContours 方法。边缘提取代码如下。

```
List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
Mat hierarchy = new Mat();
```

```
Mat mask = Mat.zeros(erodedpic.size(), CvType.CV_8UC1);

//检测的轮廓不建立等级关系
int mode = Imgproc.RETR_LIST;
Imgproc.findContours(erodedpic, contours, hierarchy, mode,
        Imgproc.CHAIN_APPROX_SIMPLE);  //返回所有轮廓组成的列表

for (int idx = 0; idx < contours.size(); idx++) {
    Rect rect = Imgproc.boundingRect(contours.get(idx));

    Mat maskROI = new Mat(mask, rect);
    maskROI.setTo(new Scalar(0, 0, 0));
    Imgproc.drawContours(mask, contours, idx,
            new Scalar(255, 255, 255), Core.FILLED);

    double r = (double) Core.countNonZero(maskROI) //得到非零像素点数
            / (rect.width * rect.height);

    if (r > .45 && (rect.height > 8 && rect.width > 8)) {
        Imgproc.rectangle(image, rect.br(), new Point(rect.br().x
                - rect.width, rect.br().y - rect.height), new Scalar(0,
                255, 0));
    }
}

String outputfile = "contours.jpg";
Imgcodecs.imwrite(outputfile, image); //输出框出字符边缘后的图像
```


使用 OpenCV 中的级联分类器实现人脸检测的代码如下。

```
//加载 dll
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
CascadeClassifier faceDetector = new CascadeClassifier(
        "haarcascade_frontalface_alt.xml"); //创建级联分类器
//待识别的图片读入到矩阵对象
Mat image = Imgcodecs.imread("info1.jpg");

MatOfRect faceDetections = new MatOfRect();  //识别结果放入矩阵数组
faceDetector.detectMultiScale(image, faceDetections);

System.out.println(String.format("Detected %s faces",
        faceDetections.toArray().length));

//把识别出的人脸用矩形框出来
for (Rect rect : faceDetections.toArray()) {
    Scalar color = new Scalar(0, 255, 0);
    Imgproc.rectangle(image, new Point(rect.x, rect.y), new Point(
```

```
                rect.x + rect.width, rect.y + rect.height), color, 3);
}

//写入到新的图像文件
String filename = "info1-FaceDetector.jpg";
System.out.println(String.format("Writing %s", filename));
Imgcodecs.imwrite(filename, image);
```

这里使用自带的模型文件，也可以使用 opencv_traincascade 自己训练出 xml 模型文件。可以使用 GPU + OpenCL 提高 OpenCV 运行速度。

7.3.2　文字区域提取

把感兴趣（ROI）的文字区域存成小图片，便于 Tesseract 识别。

```
MatOfPoint cnt = contours.get(idx);
Rect rect = Imgproc.boundingRect(cnt);
Mat cropped = new Mat(image, rect);        //切出输入图片的部分区域

String outputfile = "contourshanzi_" + idx + ".jpg";
Imgcodecs.imwrite(outputfile, cropped);        //输出包含文字区域的图像
```


发现文字块的质心，然后根据质心的大小排序。先定义质心类：

```
public class ContourROI {
    public int index; //编号
    public int momentsX; //质心

    public ContourROI(int index, int momentsX) {
        this.index = index;
        this.momentsX = momentsX;
    }
}
```

收集到数组：

```
ArrayList<ContourROI> rois = new ArrayList<ContourROI>();
for (int idx = 0; idx < contours.size(); idx++) {
    MatOfPoint cnt = contours.get(idx);
    Moments mo = Imgproc.moments(cnt);
    int cx = (int) (mo.m10 / mo.m00);       //质心

    rois.add(new ContourROI(idx, cx));
}
```

最后使用 Lambda 表达式排序：

```
rois.sort((ContourROI s1, ContourROI s2) -> {
```

```
        return s1.momentsX - s2.momentsX;
    });

    String text = TessOCR.ocr(rois); //根据排序后的顺序识别图片
```

为了验证文字区域，可以检查图片中有效点的数量。只是黑和白两种颜色的图才能通过验证，带红色的图就通不过验证。

```
    Mat imgThreshold = new Mat(); //目标图片
    //让白色的点通过验证
    Core.inRange(image, new Scalar(200, 200, 200, 0), new Scalar(255, 255,
            255, 160), imgThreshold);

    //让黑色的点通过验证，也就是在目标图片上显示成白色
    Mat output = new Mat();
    Core.bitwise_not(image, output);
    Core.inRange(output, new Scalar(200, 200, 200, 0), new Scalar(255, 255,
            255, 160), imgThreshold);

    int num = Core.countNonZero(imgThreshold); //得到有效点数
```


为了提取表格中的文字块，可以采用先腐蚀，再用 Canny 边缘检测的方法切出文字块。

```
    String inputImg = "table/inclined_text_squares_rotated.jpg";
    Mat image = Imgcodecs.imread(inputImg);

    Mat gray = new Mat();
    Imgproc.cvtColor(image, gray, Imgproc.COLOR_BGR2GRAY);

    Mat bw = new Mat();       //用来存二值化的图
    Imgproc.threshold(gray, bw, 0.0, 255.0, Imgproc.THRESH_BINARY
        | Imgproc.THRESH_OTSU);

    //变形核
    Mat morphKernel = Imgproc.getStructuringElement(Imgproc.MORPH_RECT, //矩形
        new Size(7, 10)); //宽度 7,高度 10

    //用来存腐蚀后的图像对象
    Mat erodedpic = new Mat();
    //进行腐蚀
    Imgproc.erode(bw, erodedpic, morphKernel);

    //Canny 边缘检测
    Imgproc.Canny(erodedpic, erodedpic, 300, 1000, 5, true);
```

7.3.3　纠正偏斜

通过计算最小边界框的方法得到文字的倾斜角度。

```
Mat src = Imgcodecs.imread("text.png");
Mat src_gray = new Mat();
Imgproc.cvtColor(src, src_gray, Imgproc.COLOR_BGR2GRAY);
Imgcodecs.imwrite("inclined_text_src_gray.jpg", src_gray);

Mat output = new Mat();
Core.bitwise_not(src_gray, output);
Imgcodecs.imwrite("inclined_text_output.jpg", output);

Mat points = Mat.zeros(output.size(),output.type());
Core.findNonZero(output, points);

MatOfPoint mpoints = new MatOfPoint(points);
MatOfPoint2f points2f = new MatOfPoint2f(mpoints.toArray());
RotatedRect box = Imgproc.minAreaRect(points2f); //得到倾斜矩阵

Mat src_squares = src.clone();
//计算旋转矩阵
Mat rot_mat = Imgproc.getRotationMatrix2D(box.center, box.angle, 1);
Mat rotated = new Mat();
Imgproc.warpAffine(src_squares, rotated,
            rot_mat, src_squares.size(), Imgproc.INTER_CUBIC);
Imgcodecs.imwrite("inclined_text_squares_rotated.jpg",rotated);
```


RotatedRect 的角度不能提供足够的信息来了解对象的角度，还必须使用 RotatedRect 的 size.width 和 size.height。

```
RotatedRect rotatedRect = Imgproc.minAreaRect(points2f);
double blobAngleDeg = rotatedRect.angle;
if (rotatedRect.size.width < rotatedRect.size.height) {
    blobAngleDeg = 90 + blobAngleDeg;
}
//使用调整后的角度旋转矩阵
Mat rot_mat = Imgproc.getRotationMatrix2D(rotatedRect.center, blobAngleDeg, 1);
```

为了实现白底而不是黑底，需要使用 warpAffine 函数的 borderMode 和 borderValue 参数来实现这个目标。可以将模式设置为 BORDER_CONSTANT，这样就会为边框像素（即图像外部）使用常量值，可以将该值设置为要使用的常量值（即白色），代码如下。

```
Imgproc.warpAffine(src_squares, rotated, rot_mat, src_squares.size(),
                    Imgproc.INTER_CUBIC,Core.BORDER_CONSTANT,
```

```
                                new Scalar(255, 255, 255));
```

平移：

```
MatOfPoint2f srcTri;
MatOfPoint2f dstTri;
Mat warp_mat = new Mat(2, 3, CvType.CV_32FC1);
Mat src = Imgcodecs.imread(path, Imgcodecs.IMREAD_COLOR);
Mat warp_dst = new Mat(src.rows(), src.cols() * 3, src.type());
Point[] srcPointArray = { new Point(0, 0),
        new Point(src.cols() - 1.f, 0), new Point(0, src.rows() - 1.f) };
srcTri = new MatOfPoint2f(srcPointArray);
Point[] dstPointArray = {
        new Point(src.cols(), 0),
        new Point(src.cols() * 2.0f-1.f, 0),
        new Point(src.cols(), src.rows() - 1.f) };
dstTri = new MatOfPoint2f(dstPointArray);
warp_mat = Imgproc.getAffineTransform(srcTri, dstTri);

Scalar colorScalar = new Scalar(255, 255, 255, 0); //白色背景
Imgproc.warpAffine(src, warp_dst, warp_mat, warp_dst.size(),
        Imgproc.INTER_AREA, Core.BORDER_CONSTANT, colorScalar);

Imgcodecs.imwrite("transite.jpg", warp_dst);
```


7.3.4 Linux 环境支持

可以从源代码生成 Linux 下的动态链接库 libopencv_java320.so，或者下载编译好的 libopencv_java320.so 文件。使用如下的代码支持多操作系统。

```
OSType osType = OsCheck.getOperatingSystemType();
if (osType == OsCheck.OSType.Linux) {
    libName = "libopencv_java320.so";
} else if (osType == OsCheck.OSType.Windows) {
    libName = "opencv_java320.dll";
}
System.load(new File("./lib/".concat(libName)).getAbsolutePath());
```

7.4 JavaCV

Java 调用 OpenCV 目前只能在 Windows 下运行，为了能够在 Linux 下使用 Java 调用 OpenCV 的功能，可以考虑使用 JavaCV（https://github.com/bytedeco/javacv）。使用 JavaCV 提取文字区域图片的代码如下。

```
String inputImg = "hanzi.jpg";
Mat image = opencv_imgcodecs.imread(inputImg);
float ra = (float) 1000 / Math.max(image.arrayWidth(), image.arrayHeight());
opencv_core.Size dsize = new opencv_core.Size(0, 0);
Mat output = new Mat();
opencv_imgproc.resize(image, output, dsize, ra, ra, opencv_imgproc.INTER_LINEAR);

Mat grad = new Mat();
opencv_imgproc.cvtColor(image, grad, opencv_imgproc.COLOR_BGR2GRAY);

Mat bw = new Mat();         //用来存二值化的图

opencv_imgproc.threshold(grad, bw, 0.0, 255.0, opencv_imgproc.THRESH_BINARY
        | opencv_imgproc.THRESH_OTSU);

//变形核
Mat morphKernel =
    opencv_imgproc.getStructuringElement(opencv_imgproc.MORPH_RECT, //矩形
        new opencv_core.Size(6, 8)); //宽度 7,高度 10

//用来存腐蚀后的图像对象
Mat erodedpic = new Mat();
//进行腐蚀
opencv_imgproc.erode(bw, erodedpic, morphKernel);

MatVector contours = new MatVector();
Mat hierarchy = new Mat();

Mat mask = new Mat(erodedpic.size(), opencv_core.CV_8UC1);

//检测的轮廓不建立等级关系
int mode = opencv_imgproc.RETR_LIST;
opencv_imgproc.findContours(erodedpic, contours, hierarchy, mode,
        opencv_imgproc.CHAIN_APPROX_SIMPLE);

for (int idx = 0; idx < contours.size(); idx++) {
    Rect rect = opencv_imgproc.boundingRect(contours.get(idx));

    opencv_imgproc.drawContours(mask, contours, idx,
            new opencv_core.Scalar(255, 255));

    opencv_imgproc.rectangle(image, rect.br(), new Point(rect.br().x()
            - rect.width(), rect.br().y() - rect.height()), new Scalar(0,
            255));
```

```
}

String outputfile = "contourshanzi.jpg";
opencv_imgcodecs.imwrite(outputfile, image); //输出框出字符边缘后的图像
```

识别人脸：

```
public class App {
    public static void main(String[] args) throws URISyntaxException {
        String filepath = args.length > 0 ? args[0] : Paths.get(
                App.class.getResource("/lena.png").toURI()).toString();
        faceDetect(filepath);
    }

    public static void faceDetect(String filepath) throws URISyntaxException {
        String classifierName = Paths.get(
                App.class.getResource("/haarcascade_frontalface_default.xml")
                        .toURI()).toString();
        CascadeClassifier faceDetector = new CascadeClassifier(classifierName);
        System.out.println("load " + filepath);
        Mat source = opencv_imgcodecs.imread(filepath);
        RectVector faceDetections = new RectVector();
        faceDetector.detectMultiScale(source, faceDetections);
        long numOfFaces = faceDetections.limit();
        System.out.println(numOfFaces + " faces are detected!");
        for (int i = 0; i < numOfFaces; i++) {
            Rect r = faceDetections.position(i);
            opencv_imgproc.rectangle(source, new Point(r.x(), r.y()), new Point(r.x()
                    + r.width(), r.y() + r.height()), new Scalar(0, 0, 255, 0));
        }
        opencv_imgcodecs.imwrite("faces.png", source);
        faceDetector.close();
    }
}
```


7.5 本章小结

1999 年，Intel 公司开始研发使用 C++开发的计算机视觉库 OpenCV，后来由 Itseez 公司维护。2016 年，Intel 公司收购了 Itseez。从 2.4.4 版本以后，OpenCV 开始支持 Java。直到 3.2 版本，OpenCV 的 Java API 对 Windows 以外的操作系统支持不够好。

可以采用 LeNet5 结构识别汉字。LeNet5 是最早的卷积神经网络之一，诞生于 1994 年，最初用于手写体数字识别。如果要提高精度，可以用 ResNet。

除了纠正偏斜以外，还可以进行笔画宽度归一化。

Tesseract 最初是由惠普公司在 1985 年至 1994 年期间开发的，1996 年进行了较多的更改，以便于移植到 Windows 操作系统，并在 1998 年进行了一些 C++化的改造。2005 年，惠普开放了 Tesseract 源代码。自 2006 年起，Tesseract 由 Google 公司开发。

最新的稳定版本是 3.05.01，于 2017 年 6 月 1 日发布。最新的 3.05 源代码可以从 github 的 3.05 分支获得。

Tesseract 新的基于 4.00.00alpha 版本的源代码基于长短时记忆网络（LSTM）实现，可从 github 上的主分支获得。

问答式搜索

我们应该把意见和论据套入一个类似数据库的正规架构中，如一个涵盖人类在某主题下所有知识的数据库。

在搜索结果中需要包含输入问题的答案，如“如何补办身份证”。这里首先介绍一种理解语义的方法，然后介绍数据深度整合的方法，以便能够得到更好的答案。

8.1 生成表示语义的代码

为了实现对文本的理解，可以根据文本生成对应的计算机源代码。例如，根据“猫喵喵地叫”生成 Java 源代码。使用 JCodeModel（https://github.com/phax/jcodemodel）生成代码如下。

```
public static void main(String[] args) throws IOException, JClassAlreadyExistsException {
    JCodeModel cm = new JCodeModel();
    JDefinedClass dc = cm._class("animal.Cat");

    JMethod m = dc.method(0, String.class, "say");
    m.body()._return(JExpr.lit("meow"));

    File file = new File("./target/classes");
    file.mkdirs();
    cm.build(file);
}
```

得到的源代码如下。

```
public class Cat {
    String say() {
        return "meow";
    }
}
```

也可以使用 JDT（https://github.com/eclipse/eclipse.jdt.core）生成源代码。生成源代码的

例子如下。

```
AST ast = AST.newAST(AST.JLS8);
CompilationUnit unit = ast.newCompilationUnit();
PackageDeclaration packageDeclaration = ast.newPackageDeclaration();
packageDeclaration.setName(ast.newSimpleName("example"));
unit.setPackage(packageDeclaration);
ImportDeclaration importDeclaration = ast.newImportDeclaration();
QualifiedName name = ast.newQualifiedName(ast.newSimpleName("java"),
        ast.newSimpleName("util"));
importDeclaration.setName(name);
importDeclaration.setOnDemand(true);
unit.imports().add(importDeclaration);
TypeDeclaration type = ast.newTypeDeclaration();
type.setInterface(false);
type.modifiers().add(
        ast.newModifier(Modifier.ModifierKeyword.PUBLIC_KEYWORD));
type.setName(ast.newSimpleName("HelloWorld"));
MethodDeclaration methodDeclaration = ast.newMethodDeclaration();
methodDeclaration.setConstructor(false);
List modifiers = methodDeclaration.modifiers();
modifiers.add(ast.newModifier(Modifier.ModifierKeyword.PUBLIC_KEYWORD));
modifiers.add(ast.newModifier(Modifier.ModifierKeyword.STATIC_KEYWORD));
methodDeclaration.setName(ast.newSimpleName("main"));
methodDeclaration.setReturnType2(ast
        .newPrimitiveType(PrimitiveType.VOID));
SingleVariableDeclaration variableDeclaration = ast
        .newSingleVariableDeclaration();
variableDeclaration.setType(ast.newArrayType(ast.newSimpleType(ast
        .newSimpleName("String"))));
variableDeclaration.setName(ast.newSimpleName("args"));
methodDeclaration.parameters().add(variableDeclaration);
org.eclipse.jdt.core.dom.Block block = ast.newBlock();
MethodInvocation methodInvocation = ast.newMethodInvocation();
name = ast.newQualifiedName(ast.newSimpleName("System"),
        ast.newSimpleName("out"));
methodInvocation.setExpression(name);
methodInvocation.setName(ast.newSimpleName("println"));
InfixExpression infixExpression = ast.newInfixExpression();
infixExpression.setOperator(InfixExpression.Operator.PLUS);
StringLiteral literal = ast.newStringLiteral();
literal.setLiteralValue("Hello");
infixExpression.setLeftOperand(literal);
literal = ast.newStringLiteral();
literal.setLiteralValue(" world");
infixExpression.setRightOperand(literal);
```

```
methodInvocation.arguments().add(infixExpression);
ExpressionStatement expressionStatement = ast
        .newExpressionStatement(methodInvocation);
block.statements().add(expressionStatement);
methodDeclaration.setBody(block);
type.bodyDeclarations().add(methodDeclaration);
unit.types().add(type);

System.out.println("unit: " + unit.toString());
```

输出的源代码如下。

```
package example;
import java.util.*;
public class HelloWorld {
   public static void main(    String[] args){
      System.out.println("Hello" + " world");
   }
}
```


假设本体包含两种类型的资源，即 Agent 和 Person，后者是前者的子类。可以根据本体生成 Java 类：

```
public class Agent{
}
public class Person extends Agent{
   private String name;
}
```

JavaPoet 也可以生成源代码。可以从 https://github.com/square/javapoet 下载唯一的 jar 包 javapoet-1.9.0.jar，然后在项目中增加对这个包的引用。

com.squareup.javapoet.MethodSpec 定义方法，com.squareup.javapoet.TypeSpec 定义类，com.squareup.javapoet.JavaFile 输出源代码，生成代码如下。

```
//创建方法
MethodSpec main = MethodSpec
        .methodBuilder("say")//方法名 say
        .addModifiers(Modifier.PUBLIC, Modifier.STATIC)//方法名前的修饰关键字
        .returns(void.class)
        .addStatement("$T.out.println($S)", System.class,
                "meow").build();//这里的$T 和$S 都是占位符

//创建类
TypeSpec helloWorld = TypeSpec.classBuilder("Cat")//类名 Cat
        .addModifiers(Modifier.PUBLIC)
        .addMethod(main) //在类中添加方法
        .build();
```

```
//创建 java 文件
JavaFile javaFile = JavaFile.builder("com.example.helloworld",
        helloWorld).build();

javaFile.writeTo(System.out);
```

生成如下的 Java 代码。

```
package com.example.helloworld;

import java.lang.System;

public class Cat {
   public static void say() {
      System.out.println("meow");
   }
}
```


RDFReactor（https://github.com/semweb4j/semweb4j）工作在 RDF2go 之上。RDF2go 是一个抽象层，实现了 Jena 或 Sesame。

代码生成可以通过如下的一行代码完成：

```
CodeGenerator.generate("bucket.owl", "src", "ontotest.model1", Reasoning.rdfs, true, true);
```

也可以使用 Byte Buddy（https://github.com/raphw/byte-buddy）在 Java 代码中即时生成代码。

可以把预设知识也表示成代码的形式。从维基百科提取结构化信息并组织成本体形式的 DBpedia 就可以作为一个预设知识库。DBpedia 信息提取框架项目的链接是https://github.com/dbpedia/extraction-framework/。

使用 JDT 编译代码。

```
IProject myProject;
IProgressMonitor myProgressMonitor;
myProject.build(IncrementalProjectBuilder.INCREMENTAL_BUILD, myProgressMonitor);
```

或者使用虚拟机自带的编译器 JavaCompiler 编译源代码。

```
//先检查 java.home/lib 下的 tools.jar 文件，这个文件必须存在
System.out.println(System.getProperty("java.home"));
JavaCompiler compiler=ToolProvider.getSystemJavaCompiler();

System.out.println("JavaCompiler: " + compiler);
int results = compiler.run(null, null, null, "f:/test/Cat.java");
System.out.println((results == 0) ? "编译成功" : "编译失败");
```

可以使用 AspectJ（https://github.com/eclipse/org.aspectj）监测对某个方法的调用。

如果游泳对应 Person 类的一个方法 swmming()，则用 AspectJ 监测对这个方法的调用情况。

例如，根据文本“小明昨天去游泳了”生成代码：

```
xiaoming.swmming();
```

为了回答问题“谁去游泳了”。可以执行根据文本生成的代码，监测到 xiaoming 调用了这个方法。

可以使用 JavaParser（https://github.com/javaparser）修改源代码，插入日志代码，实现对方法调用的监测；可以使用 MongoDB 存储日志信息。

创建一个项目引用 javaparser-core-3.3.2.jar 这个包。Java 源代码表示成表达式 Expression 类。所有的方法调用都表示成了 MethodCallExpr 的子类。输出源代码中所有的方法。

```
import com.github.javaparser.JavaParser;
import com.github.javaparser.ast.expr.MethodCallExpr;
import com.github.javaparser.ast.visitor.VoidVisitorAdapter;
import com.google.common.base.Strings;

import java.io.File;
import java.io.IOException;

public class MethodCallsExample {
    public static void listMethodCalls(File projectDir) {
        new DirExplorer((level, path, file) -> path.endsWith(".java"), (level,
                path, file) -> {
            System.out.println(path);
            System.out.println(Strings.repeat("=", path.length()));
            try {
                new VoidVisitorAdapter<Object>() {
                    @Override
                    public void visit(MethodCallExpr n, Object arg) {
                        super.visit(n, arg);
                        System.out.println(" [L " + n.getBegin().get().line
                                + "] " + n);
                    }
                }.visit(JavaParser.parse(file), null);
                System.out.println(); //空行
            } catch (IOException e) {
                new RuntimeException(e);
            }
        }).explore(projectDir);
    }

    public static void main(String[] args) {
        String dir = "F:/workspace/JavaParserPlay/src/example";
```

```
            File projectDir = new File(dir);
            listMethodCalls(projectDir);
        }
    }
```

8.2 信息整合

为了得到更完整的知识，需要整合不同来源的信息。连接多个知识库中的等价实体称为实体对齐。本节介绍实体对齐的实现方法。

8.2.1 实体对齐

Duke（https://github.com/larsga/duke）可用于在单个表/数据源中查找重复记录，也可用于查找最有可能代表相同现实世界实体的不同表/源中的记录。Duke 从指定的数据源加载数据，然后通过调用清理器（Cleaner）清除数据（即归一化）。

有两种操作模式：DeduplicationMode 和 RecordLinkageMode。

8.2.2 编辑距离

考虑如何衡量两个字符串 S 和 T 之间的差异 distance(S, T)。如 gold 和 good 之间只差一个字母，而 gold 和 bad 之间差更多的字母，所以 distance(gold, good)< distance(gold, bad)。

给定两个字符串：$S = s_1 s_2 \cdots s_m$ 和 $T = t_1 t_2 \cdots t_n$

假设一个打字员，把单词 t 错误地输入成为单词 s 了，她需要若干次操作来改正这个错误。对于把 S 转换到 T 所需要的一系列编辑操作，想要找到最小的花费 $D(S,T)$。编辑操作包括以下三种：

（1）使用 T 中的一个字符替换 S 中的一个字符；

（2）删除 S 中的一个字符；

（3）插入 T 中的一个字符。

$D(S,T)$称为编辑距离。$D(S,T)$是一个不小于零的整数。例如，把 gold 转换成 good 最少需要 1 次编辑操作，也就是把 l 替换成 o，所以 D(gold, good)=1。而把 gold 转换成 bad 最少需要 3 次编辑操作，所以 D(gold, bad)=3。因此，D(gold, good)< D(gold, bad)。

distance(gold, good)可以看成这样计算得到的：gold 的最后一个字符 d 替换成 good 中的 d。因为这两个字符相同，所以 d 这个字符并没有导致编辑距离增加，代价是 0。

对于 S 中的字符 s 和 T 中的字符 t 来说，假设 t 是由字符 s 替换得到的。因为 t 的存在，导致编辑距离增加的值称为代价 c。根据三角形中两边长度之和大于第三条边。对于代价 c，假设三角不等式：

$$c(a, c) \leqslant c(a, b) + c(b,c)$$

从 a 转换成 c 的代价不大于要通过 b 这个步骤的代价。$c(a, c)$约减成两个字符之间的变换。也就是说，每个字符最多改变一次。与之类似，日常生活中往往不通过中介而是找房东直接租房成本要低。

如果 s 和 t 相同，则代价是 0。如果 s 和 t 不同，则代价是 1。代价模型用伪代码表示：

```
if(s!=t) c(s,t)=1
if(s==t) c(s,t)=0
```

distance(goods, good)可以看成这样计算得到的：源串删除最后一个源字符 s 就得到了目标字符串 good。

distance(goo, good)可以看成这样计算得到的：源串插入最后一个目标字符 d 就得到了目标字符串。

假设把 S 的子串 $s_1 \cdots s_i$ 写作 S_i。而 T 的子串 $t_1 \cdots t_j$ 写作 T_j。$D_{i,j} = D(S_i, T_j)$

把 S 转换到 T 有三种结束的可能：

- 最后一个字符是从源字符替换成目标字符。如果源字符串和目标字符串的最后一个字符正好相等，则代价是 0，否则代价是 1。用 t_n 替换 s_m：$D_{m,n} = D_{m-1,n-1} + c(s_m, t_n)$。
- 删除最后一个源字符就得到了目标字符串。删除 s_m：$D_{m,n} = D_{m-1,n} + 1$。
- 插入最后一个目标字符就得到了目标字符串。插入 t_n：$D_{m,n} = D_{m,n-1} + 1$。

$D_{i,j}$ 由 $D_{i-1,j-1}$、$D_{i-1,j}$、$D_{i,j-1}$ 推出。计算 $D_{i,j}$ 如图 8-1 所示。

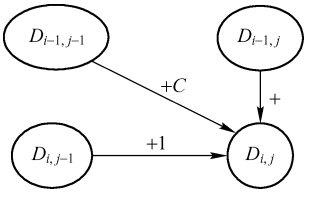

图 8-1　计算编辑距离问题分解图

把“levenshtein”转换成“meilenstein”的过程如图 8-2 所示。

```
l  e     v  e  n  s  h  t  e  i  n
o  =  +  o  =  =  =  -  =  =  =  =
m  e  i  l  e  n  s     t  e  i  n
```

图 8-2　"meilenstein"和"levenshtein"的编辑距离

在图 8-2 中，“=”表示匹配上了；“o”表示替换；“+”表示插入；“-”表示删除。

形式化的写法：

$D_{m-1,n}$ -> $D_{m,n}$ 就是 -s

$D_{m-1,n-1}$ -> $D_{m,n}$ 就是 s o t

$D_{m,n-1}$ ->$D_{m,n}$ 就是 +t

如果 D(gol, goo)已知，则 D(gold, good)也就知道了，因为 D(gold, good)和 D(gol, goo)的值相同。所以可以认为 D(gold, good)依赖于 D(gol, goo)的值。而 D(gol, goo)又依赖于 D(go, go)的值，D(gol, goo)比 D(go, go)的值大一个。

一个问题可以由子问题推出。图 8-3 是计算 D(abc,abb)的问题分解图。其中 D(abc,abb)依赖于 D(abc,ab)、D(ab,abb)和 D(ab,ab)的结果。D(abc,ab) -> D(abc,abb)就是目标单词 T 增加一个字符 b。D(ab,abb) -> D(abc,abb)就是源单词 S 删除一个字符 c。D(ab,ab) -> D(abc,abb)就是源单词 S 最后一个字符 c 替换成 b。这里的 D(ab,ab)在图 8-3 中重复出现了，也就是说分

解计算 *D*(abc,abb)时，会重复计算 *D*(ab,ab)。

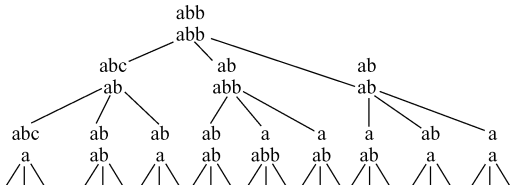

图 8-3　计算编辑距离问题分解图

编辑距离就是从源串转换成目标串的最小代价，所以取最小值，走近路。就像出租车在一般情况下不能绕远路，否则这样车费就高了。

求解编辑距离的循环等式（Recurrence Equation）：

$$D_{m,n} = \min \begin{Bmatrix} D_{m-1,n-1} & + & c(s_m, t_n), \\ D_{m-1,n} & + & 1, \\ D_{m,n-1} & + & 1 \end{Bmatrix}$$

需要计算所有的 $D_{i,j}$ 当 $0 \leqslant i \leqslant m, 0 \leqslant j \leqslant n$ 时。

分治法的实现方法采用自顶向下递归求解，代码如下。


```
public class RecursiveEditDistance {
    private String s; //源串
    private String t; //目标串

    public int distance(final String s1, final String s2) {
        this.s = s1;
        this.t = s2;
        //整个问题是: 对于 i = |S| 并且 j = |T| ，计算 D(i,j)
        return distance(s1.length(), s2.length());
    }

    // s 中的前 i 个字符到 t 中的前 j 个字符之间的距离
    protected int distance(final int i, final int j) {
        //去掉下面这行代码的注释符就能看到重复计算子问题
        // System.out.println(String.format("Checking pair: %s, %s", i, j));
        if (i == 0) { //如果源字符串长度是 0，则编辑距离就是目标字符串的长度
            return j;
        }
        if (j == 0) { //如果目标字符串长度是 0，则编辑距离就是源字符串的长度
            return i;
        }
        //循环等式
        int cost = 0;
        if (s.charAt(i - 1) != t.charAt(j - 1)) {
            cost = 1;
        }
```

```
            //把当前问题分解成 3 个子问题
            int del = distance(i - 1, j) + 1;          //源串删除一个字符成为目标串
            int ins = distance(i, j - 1) + 1;          //源串插入一个字符成为目标串
            int rep = distance(i - 1, j - 1) + cost; //源串替换一个字符成为目标串
            return Math.min(del, Math.min(ins, rep));
        }
    }
```

如果采用分治法，则有些子问题的计算是重复的。所以要把子问题的计算结果存起来。动态规划方法采用从底向上计算。通用的方法是用 HashMap 存储计算的中间结果，这里采用二维数组。实现代码如下。

```
    /**
     * @param s  输入源串
     * @param t  输入目标串
     * @return  源串和目标串之间的编辑距离
     */
    public static int editDistance(String s, String t) {
        int i; //遍历源字符串的变量
        int j; //遍历目标字符串的变量
        char s_i; //源字符串的第 i 个字符
        char t_j; //目标字符串的第 j 个字符
        int cost; //代价

        //第 1 步，初始化
        int n = s.length(); //源字符串的长度
        int m = t.length(); //目标字符串的长度
        if (n == 0) {
            return m;
        }
        if (m == 0) {
            return n;
        }
        int[][] d = new int[n + 1][m + 1]; //最小花费矩阵

        //第 2 步，初始化第一行成为 0…n
        for (i = 0; i <= n; i++) {
            d[i][0] = i;
        }

        //初始化第一列成为 0…m
        for (j = 0; j <= m; j++) {
            d[0][j] = j;
        }
```

```
        //第 3 步，检查字符串 s 的每个字符(i 为 1～n)。
        for (i = 1; i <= n; i++) {
            s_i = s.charAt(i - 1);

            //第 4 步，检查字符串 t 中的每个字符 (j 为 1～m)。
            for (j = 1; j <= m; j++) {
                t_j = t.charAt(j - 1);

                //第 5 步
                //如果 s[i]等于 t[j]，则代价是 0
                //如果 s[i]不等于 t[j]，则代价是 1
                if (s_i == t_j) {
                    cost = 0;
                } else {
                    cost = 1;
                }

                //第 6 步
                //设置矩阵的单元 d[i,j]等于下面 3 个数的最小值
                //a. 上面的单元加 1: d[i-1,j] + 1
                //b. 左边的单元加 1: d[i,j-1] + 1
                //c. 左上对角的单元加代价: d[i-1,j-1] + cost
                d[i][j] = Minimum(d[i - 1][j] + 1, d[i][j - 1] + 1,
                        d[i - 1][j - 1] + cost);
            }
        }

        //第 7 步
        //在迭代步骤(3, 4, 5, 6)结束后，距离在单元 d[n,m]中
        return d[n][m];
    }
```


计算两个大字符串的编辑距离会导致内存浪费太多而出现溢出现象。其实只要计算当前这一步需要的中间结果，不需要保留更早的结果。所以可以用一维数组替代二维数组。维护两个一维数组，长度是 s.length()+1。当遍历字符串 s 的字符时，第 1 个一维数组 d 是“当前工作”的距离数组，记录了最新的距离代价计数。每次增加正在比较的目标字符串 t 的索引时，把 d 复制到第 2 个一维数组，把这个数组称为 p。这样做能保留以前的代价计数，因为算法需要这样。计算 3 个数的最小值。注意，数组并不是真的复制了，仅仅是交换了。这显然远好于每次通过外循环复制一个数组或做一个 System.arraycopy()。

```
    public static int editDistance(String s, String t) {
        if (s == null || t == null) {
            throw new IllegalArgumentException("字符串不能是空");
        }

        int n = s.length(); //源字符串的长度
```

```
        int m = t.length(); //目标字符串的长度

        if (n == 0) {
            return m;
        } else if (m == 0) {
            return n;
        }

        int p[] = new int[n + 1]; //前一个水平的代价数组
        int d[] = new int[n + 1]; //水平的代价数组
        int _d[]; //用来帮助交换 p 和 d 的占位符

        //字符串 s 和 t 的索引
        int i; //遍历字符串 s
        int j; //遍历字符串 t

        char t_j; // t 的第 j 个字符

        int cost; //代价

        for (i = 0; i <= n; i++) {
            p[i] = i;
        }

        for (j = 1; j <= m; j++) {
            t_j = t.charAt(j - 1);
            d[0] = j;

            for (i = 1; i <= n; i++) {
                cost = s.charAt(i - 1) == t_j ? 0 : 1;
                // left+1 和 top+1，以及左上对角+cost 单元的最小值
                d[i] = Math.min(Math.min(d[i - 1] + 1, p[i] + 1), p[i - 1]
                        + cost);
            }

            //把当前距离计数器行变成前一行的距离计数器
            _d = p; //前一行计数器没用了，重用作当前距离计数器行
            p = d;
            d = _d;
        }

        //上面循环中的最后一次动作交换了 d 和 p，因此 p 现在包含最近的代价计数
        return p[n];
    }
```

这个实现和上一个实现的区别是：当这个实现计算两个非常大的字符串时，也不会导

致内存溢出。

两个字符交换也算一次编辑操作。例如，ab 只需要交换一次位置就可以转换成 ba，所以 ab 和 ba 的编辑距离是 1。增加交换位置的距离。

```
int swapDist;
if(s1.length() > 1 && s2.length() > 1 &&
      s1.charAt(0) == s2.charAt(1) && s1.charAt(1) == s2.charAt(0))
   swapDist = editDist(s1.substring(2), s2.substring(2)) + 1;
else
   swapDist = Integer.MAX_VALUE;   //如果前两个字符不匹配，就不能交换
```

8.2.3　Jaro-Winkler 距离

对于专有名词可以使用 Jaro-Winkler 距离。因为假设首字母不容易拼错，所以 Jaro-Winkler 给予了起始部分就相同的字符串更高的分数。

```
public class Jaro {
      /**
       * 得到两个字符串的 Jaro 距离相似度
       *
       * @param string1 第 1 个输入字符串
       * @param string2 第 2 个输入字符串
       * @return 一个 0～1 之间的值
       */
      public float getSimilarity(final String string1, final String string2) {
            //得到字符串长度的一半（用于可接受的换位的距离）
            final int halflen = ((Math.min(string1.length(), string2.length())) / 2) + ((Math. min(string1.length(),
string2.length())) % 2);

            //得到公共字符串
            final StringBuffer common1 = getCommonCharacters(string1, string2, halflen);
            final StringBuffer common2 = getCommonCharacters(string2, string1, halflen);

            //检查没有共同字符的情况
            if (common1.length()== 0 || common2.length()== 0) {
                  return 0.0f;
            }

            //检查同样长度的公共字符串，如果不相同，则返回 0.0f
            if (common1.length() != common2.length()) {
                  return 0.0f;
            }

            //得到换位的数量
            int transpositions = 0;
```

```
                int n=common1.length();
                for (int i = 0; i < n; i++) {
                    if (common1.charAt(i) != common2.charAt(i))
                        transpositions++;
                }
                transpositions /= 2.0f;

                //计算 Jaro 指标
                return (common1.length() / ((float) string1.length()) +
                        common2.length() / ((float) string2.length()) +
                        (common1.length() - transpositions) / ((float) common1.length())) / 3.0f;
            }

            /**
             *返回一个 string2 内的从 string1 开始的字符串缓存，如果只有给定距离的间隔
             *
             * @param string1
             * @param string2
             * @param distanceSep
             * @return 返回一个字符串缓存
             */
            private static StringBuffer getCommonCharacters(final String string1, final String string2, final int distanceSep) {
                //创建字符的返回缓冲区
                final StringBuffer returnCommons = new StringBuffer();
                //创建一个 string2 的副本用于处理
                final StringBuffer copy = new StringBuffer(string2);
                //迭代 string1
                int n=string1.length();
                int m=string2.length();
                for (int i = 0; i < n; i++) {
                    final char ch = string1.charAt(i);
                    //设置布尔值用于查找，如果找到，则快速退出循环
                    boolean foundIt = false;
                    //将字符与两边的字符范围进行比较
                    for (int j = Math.max(0, i - distanceSep); !foundIt && j < Math.min(i + distanceSep, m - 1); j++) {
                        //检查是否找到
                        if (copy.charAt(j) == ch) {
                            foundIt = true;
                            //附加发现的字符
                            returnCommons.append(ch);
                            //更改复制的 string2 用于接下来的处理
                            copy.setCharAt(j, (char)0);
                        }
                    }
```

```
            }
            return returnCommons;
        }
    }
```

计算两个词的 Jaro-Winkler 相似度：

```
String s1 ="sony";
String s2 ="sonf";
Jaro j = new Jaro();
System.out.println(j.getSimilarity(s1, s2));
```

8.2.4　比较器

比较器可以比较两个字符串值，并产生 0.0（意义完全不同）和 1.0（意为完全相等）之间的相似性度量。使用相似性度量比只是知道两个值是否相同更好一些。而且，不同类型的值比较的方法不一样。例如，比较名称和地址等复杂的字符串本身就是一门学问。

Duke 提供了一些比较器，并且可以通过 no.priv.garshol.duke.Comparator 接口来实现自己的比较器。

- Levenshtein

使用 Levenshtein 编辑距离来计算相似度。基本上，它测量从字符串 1 到字符串 2 所需的编辑操作的数量。

- WeightedLevenshtein

WeightedLevenshtein 是 Levenshtein 的可配置版本，其中编辑操作可以分配不同的权重。在数字组成字符串的一部分的情况下非常有用，因为数字的差异比字母的差异更重要。地址就是其中的一个例子，因为“Main Street 12”和“Main Street 14”有很大的不同。

- Jaro-Winkler

Jaro-Winkler 相似性度量，是英文短文本去重的最佳可用通用串比较器。可以使用它计算名字和姓氏的相似度。对于一般的长字符串效果不太好。

- QGramComparator

QGramComparator 看起来和 Levenshtein 相似，但它并不在乎词的顺序，所以对于其中的词可能重新排序的字符串，效果比 Levenshtein 要好。

8.2.5　Cleaner

清理器的工作是通过从数据值中删除所有不太可能表示真正差异的变体来让比较数据变得容易。例如，一个清理器可以从邮政编码中剥离除数字以外的其他东西，或者它能规范化和小写化地址，或者将日期翻译成通用格式。

Duke 提供了一些清理器。通过实现 no.priv.garshol.duke.Cleaner 接口也可以很容易自己实现一个清理器。如果要使用正则表达式清理数据，可以写个 AbstractRuleBasedCleaner 的子类。

有如下几个基本的字符串清理器。

- LowerCaseNormalizeCleaner

最广泛使用的字符串值清理器。它将所有字母小写化，清除开始和结束位置的空格，并在 Token 之间规范化空格。 它也删除口音，将é转换成 e 等。

- DigitsOnlyCleaner

此清理器会删除不是数字的所有内容，并可用于清理邮政编码，如上所述。

- TrimCleaner

只在输入字符串的开始和结尾处去掉空格。

此外，还有些可配置的清理器和解析内容的清理器，如把挪威公司名称“a/s”规一化成为“as”等。

8.2.6 运行过程

假设我们尝试识别客户表中重复的客户记录。 假设表中有 3 行，如表 8-1 所示。

表 8-1 客户记录

ID	NAME	ADDRESS	ZIP	EMAIL
1	J. Random Hacker	Main St 101	21231	
2	John Random Hacker	Mian Street 101	21231	hack@gmail.com
3	Jacob Hacker	Main Street 201	38122	jacob@hotmail.com

清理后的数据如表 8-2 所示。

表 8-2 清理后的数据

ID	NAME	ADDRESS	ZIP	EMAIL
1	j. random hacker	main street 101	21231	
2	john random hacker	mian street 101	21231	hack@gmail.com
3	jacob hacker	main street 201	38122	jacob@hotmail.com

配置文件如下。

```
<schema>
  <threshold>0.732</threshold>

  <property type="id">
    <name>ID</name>
  </property>
  <property>
    <name>NAME</name>
    <comparator>no.priv.garshol.duke.comparators.QGramComparator</comparator>
    <low>0.35</low>
    <high>0.88</high>
```

```
    </property>
    <property>
      <name>ADDRESS1</name>
      <comparator>address-comp</comparator>
      <low>0.25</low>
      <high>0.65</high>
    </property>
    <property>
      <name>EMAIL</name>
      <comparator>no.priv.garshol.duke.comparators.ExactComparator</comparator>
      <low>0.4</low>
      <high>0.8</high>
    </property>
    <property>
      <name>ZIP</name>
      <comparator>no.priv.garshol.duke.comparators.ExactComparator</comparator>
      <low>0.45</low>
      <high>0.6</high>
    </property>
</schema>
```


阈值表示，除非记录代表同一事物的概率为 0.732（即 73.2%）或更高，否则不会将记录视为相同的。如果属性值完全相同或完全不同，则高和低元素给出使用的概率。

记录 1 和记录 2 的比较过程如下。

```
---ADDRESS1
'main street 101' ~ 'mian street 101': 0.867 (prob 0.6127)
Result: 0.5 -> 0.6127

---NAME
'j. random hacker' ~ 'john random hacker': 0.8 (prob 0.78542)
Result: 0.6127 -> 0.8527

---ZIP
'21231' ~ '21231': 1.0 (prob 0.6)
Result: 0.8527 -> 0.8967

Overall: 0.8967
```

首先比较相似的地址，相似度为 0.87。 这使我们有 0.61 的记录是相同的概率。 然后我们比较名称，这也是很相似的，所以这个属性的概率是 0.79。我们再将其与来自地址（0.61）的概率相结合，得到 0.85 的新概率。邮政编码也是一样的，所以得到更高的概率，最后以 0.897 的概率超过 0.732，所以 Duke 认为这两条记录代表同一个人。

8.2.7 遗传算法调整参数

创建一个准确的配置可能很棘手，特别当你是 Duke 新手时。为了帮助解决这个问题，Duke 提供了一种可以自动调整配置的遗传算法。这可能听起来像魔术，但它实际上是有效的。

请注意，遗传算法根本不了解 Duke，因此可能生成相当奇怪的配置。也就是说，一般情况下出来的配置会起作用，但它也可能看起来没有意义，并且可能难以手动调节。

运行遗传算法有以下两种方法。

- 被动模式要求有包含数据集中所有正确匹配项的测试文件。如果这很难达到，可以将数据的一部分用于训练目的，并为此创建一个测试文件。
- 主动模式主动询问匹配是否正确，所以无须测试文件。与被动模式相比，它只有较少的信息用来处理，但通过仅寻求最具信息性的潜在匹配来尝试弥补。在默认情况下，每一代它会向你询问 10 个问题，但你可以手动调整问题数。

如果可以使用被动模式，这意味着你可以让遗传算法无人值守地运行，并产生更好的结果。

如果想看到遗传算法的实例，可以用 Duke 附带的数据集尝试它。要尝试使用 LinkingCountries 示例，只需运行：

```
java no.priv.garshol.duke.genetic.Driver countries.xml
```

首先需要一个完整的 Duke 配置。也就是说，必须有数据源，清理和比较器设置。阈值、比较器和高低概率不一定有任何意义，但必须设置它们。让 Duke 知道模式中有哪些属性。

如果没有测试文件，只需运行遗传算法：

```
java no.priv.garshol.duke.genetic.Driver --output=genetic.xml config.xml
```

这将以主动学习模式运行，并在每一代后将最佳配置输出到 genetic.xml。可以随时停止进程，并尝试得到的配置。

请注意，使用这种方法，遗传算法会忘记所有的答案，所以如果再次运行，可能需要再次回答相同的问题。为了避免这种情况，可以这样运行它：

```
java no.priv.garshol.duke.genetic.Driver --output=genetic.xml --linkfile=links.txt config.xml
```

现在所有的答案都将被记录在 links.txt 中。如果停止算法并想再次运行，可以运行：

```
java no.priv.garshol.duke.genetic.Driver --output=genetic.xml --linkfile=links.txt --testfile=links.txt --active config.xml
```

现在它会记住上次告诉它的，但它仍然会问你每一代的 10 个问题。 它还将假设存在不在测试文件中的正确链接。任何新的答案将添加到链接文件的末尾，所以可以用这些参数重新运行多次。

如果有一个测试文件，只需这样运行 Duke：

```
java no.priv.garshol.duke.genetic.Driver --output=genetic.xml --testfile=links.txt config.xml
```

用了这些参数后，Duke 将不会询问任何问题，并且将持续运行，假设所有正确的链接都在测试文件中，并且所有不在其中的链接都是错误的。

8.3 自动问答

可以用爬虫抓取问题，然后替换其中的关键词为通用的类别，就能得到问句模板。例如,问句“糖尿病怎么治疗”替换其中的“糖尿病”为疾病名，得到问句模板“<疾病>怎么治疗”。把这样的方法称为泛化。例如，问句“张三 1999 年到 2002 年 11 月任职什么”泛化成“<人名><开始时间>到<结束时间>任职什么”。

答案句也可以做泛化处理，例如，“多功能监控表的宽范围交直流通用电源是多少”对应的答案句“宽范围交直流通用电源：AC/DC 80～270V”，能泛化成：“<num>～<num>V”。

8.3.1 问句处理器

不同的问句需要不同的处理方式。例如，“1 加 1”和“今天天气怎么样”需要不同的处理器。有很多个处理器同时匹配一个问句。现有的一些处理器包括：处理简单问答对用的 ChatHandler、处理加法的 AddHandler、处理英文单词的 WordHandler 等。

处理器从问句提取关键词，提取出来的关键词以键/值对的形式存入 PairListString 对象。

```
//键可以重复
public class PairListString {
    String[] values;
    int count;

    public PairListString(int initialCapacity) {
        values = new String[initialCapacity * 2];
    }

    /**
     * 增加一个名称/值对
     *
     * @param x  名称
     * @param y  名称对应的值
     */
    public void addPair(String x, String y) {
        if (count * 2 >= values.length) {
            values = Arrays.copyOf(values, values.length * 2);
        }
        values[count * 2] = x;
        values[count * 2 + 1] = y;
```

```
            count++;
        }

        public String getX(int index) {
            return values[index * 2];
        }

        public String getY(int index) {
            return values[index * 2 + 1];
        }

        public String get(String key){
            for (int i = 0; i < count; ++i) {
                if(values[i * 2].equals(key)){
                    return values[i * 2 + 1];
                }
            }

            return null;
        }
    }
```

使用这个类存放从问句“糖尿病怎么治疗”中提取的参数，示例代码如下。

```
PairListString questionArgs = new PairListString(1);

String type = "DiseaseName";                    //键
String diseaseName = "糖尿病";                  //值

questionArgs.addPair(type, diseaseName);
System.out.println(questionArgs.get(type));    //输出糖尿病
```

动态加载问句处理器。例如：

```
String handleClass = "questionHandler."+handleName+"Handler";//得到类名
Class<? extends QuestionHandler> clz = Class.forName(
        handleClass).asSubclass(QuestionHandler.class);
QuestionHandler answer = clz.newInstance();
String ans = answer.getAnswer(kb,g.questionArgs);
```

处理问句的规则：

```
String ruleStr = "<num>{plusnum1}加<num>{plusnum2}";
ArrayList<RuleToken> tokens = IERuleParser.getSeq(ruleStr,67); //规则以及编号
for (RuleToken t : tokens) {
    System.out.println(t);
}
```

```
Rule rule = RuleBuilder.create(tokens); //根据 Token 序列创建规则
System.out.println(rule);
System.out.println("is valid:"+rule.valid());
```

根据问句模板处理问句：

```
QuestionGrammar g = new QuestionGrammar();

String right = "<Begin><num>{plusnum1}加<num>{plusnum2}<End>";
g.add("Add", right); //处理器名称:Add

right = "<Begin><DiseaseName>{disease}怎么治疗<End>";
g.add("Cure", right); //处理器名称:Cure

String type = "DiseaseName";
String diseaseName = "糖尿病";
//仅增加词，而不是加规则
g.addWord(diseaseName, type);

TextExtractor ie = new TextExtractor(g);

String question = "糖尿病怎么治疗";
AdjList adjList = ie.getLattice(question);
System.out.println(adjList);
```

根据问句模板返回答案：

```
KnowlegeBase kb = new KBSqlite(); //使用 Sqlite 存储知识库
GrammarAnswer answer = new GrammarAnswer(kb);

String question = "糖尿病怎么治疗";
String ans = answer.getAnswer(question);

System.out.println(ans);
```

TextExtractor 类根据问句模板匹配问句：

```
public class TextExtractor{
    public TernarySearchTrie dic;      //基本词和对应的类型
    public Trie rule;                  //文法树
    FSTSGraph fstSeg;                  //用于问句原子切分
}
```

根据基本词形成切分词图：

```
//输入问句，返回切分词图
public AdjList getLattice(String text) throws Exception {
    int sLen = text.length();//字符串长度
```

```
SegScheme segScheme = fstSeg.seg(text);
AdjList g = segScheme.wordGraph; //增加虚拟的开始和结束节点,用于需要完全匹配的提取表达式

ArrayList<WordEntry> wordMatch = new ArrayList<WordEntry>();
StringBuilder unknowBuffer = new StringBuilder(); //未知词缓存
//已经处理的最大位置
int maxEnd = 0;
//生成切分词图
for (int i = 0; i < sLen; ++i) {
    boolean match = dic.matchAll(text, i, wordMatch);//到词典中查询

    if (match)//已经匹配上
    {
        for (WordEntry word : wordMatch) {
            int end = i + word.word.length();//词的开始位置
            if (end > maxEnd) {
                maxEnd = end;
            }
            g.addEdge(new CnToken(i, end, word.word, word.types));
        }
        if (unknowBuffer.length() > 0) {
            //增加未知词
            String word = unknowBuffer.toString();
            int start = i - word.length();
            HashSet<String> wordTypes = new HashSet<String>();
            wordTypes.add(TernarySearchTrie.UNKNOW_TYPE);
            if (start > maxEnd) {
                maxEnd = start;
            }
            g.addEdge(new CnToken(start, i, word, wordTypes));
            unknowBuffer.setLength(0); //重置缓存
        }
    } else if (i >= maxEnd) {
        unknowBuffer.append(text.charAt(i));
    }
}
//处理句尾没有匹配到的词
if (unknowBuffer.length() > 0) {
    //增加未知词
    String word = unknowBuffer.toString();
    int start = text.length() - word.length();

    HashSet<String> wordTypes = new HashSet<String>();
    wordTypes.add(TernarySearchTrie.UNKNOW_TYPE);
```

```
            g.addEdge(new CnToken(start, text.length(), word, wordTypes));
            unknowBuffer.setLength(0);          //重置缓存
        }

        return scanGraph(segScheme, text);      //匹配文法树
    }
```

匹配文法树的过程：

```
public AdjList scanGraph(SegScheme segScheme, String text){
    int offset = 0;//用来控制匹配的起始位置的变量
    while (offset >= 0) { //如果 i 小于此句话的长度就进入循环
        //词图和文法树做交集
        MatchSeq match = GraphMatcher.intersect(segScheme.wordGraph, offset,
                                    rule);
        if (match == null) {
            offset = segScheme.startPoints.nextSetBit(offset + 1);
            continue;
        }
        for (int i = 0; i < match.posSeq.Count; ++i) {
            NodeType n = match.posSeq[i];

            if (segScheme.endPoints.fastGet(n.end) && n.toExtract) { //检查是有效的可结束点
                String term = text.Substring(n.start, n.end - n.start);
                CnToken newEdge = new CnToken(n.start, n.end,
                                              term, n.type, false);
                segScheme.wordGraph.addNew(newEdge);
                segScheme.wordGraph.addArgs(n.type, term);
            }
        }
        //增加总的边
        NodeType nodeType = match.posSeq[match.posSeq.Count - 1];
        int end = nodeType.end;
        if (end > text.Length) {
            end = end - 1;
        }
        int start = offset;
        String termText = text.Substring(start, end - start);
        CnToken tempEdge = new CnToken(start, end,
                                       termText, match.nt, true);
        segScheme.wordGraph.addNew(tempEdge);
        offset = segScheme.startPoints.nextSetBit(offset + 1);
    }
    return segScheme.wordGraph;
}
```

8.3.2 自动发现答案

根据问题类型和标注出的正确答案寻找答案词的标注类型。

给出一个问题和相应的文档，从该文档中选择一个或多个句子作为答案。

例如，纸擦笔主要用来做什么？这个问题对应的文档是：

纸笔就叫纸擦笔，也叫卷纸擦笔。用于面积细的地方，在排好线的地方，拖出衣服的纹理及花纹，也可以在画面做出朦胧、柔润的效果，同时还可以当画笔用，当纸擦笔沾有颜色时，就利用它放在适当的地方作为阴影或暗的地方。

选择文档中的第 2 个句子作为答案句，当然答案也可以是文档中的几个句子。

先把问题分类。用包含疑问词的模板匹配出问题类型，然后对答案句分类。对答案分词和做词性标注后，也进行分类。

问题有如下这样的类型。询问人的：谁发现了北美洲？询问时间的：人类哪年登陆月球？询问数量的：珠穆朗玛峰有多高？询问定义的：什么是氨基酸？询问地点和位置的：芙蓉江在重庆市哪个县？询问原因的：天为什么是蓝的？询问一个集合的：三星手机有哪些型号？

问题类型及相关举例列表如表 8-3 所示。

表 8-3　问题类型及相关举例

问 题 类 型	对应的疑问词	例　　子
询问原因	为什么、为何、通用疑问词+（原因）	为什么天空总是蓝色的 电影《阿凡达》取得巨大的成功的原因是什么
询问时间	何时、通用疑问词+（时间 / 时候 / 年 / 月 / 日 / 天）	何时去黄山 什么时候举办上海世博会
询问人物	谁、通用疑问词+（人 / 表示人职业的名词）	谁是篮球史上最伟大的球员 哪位科学家发现了光电效应
询问地点	哪里、哪个、何处、何地、通用疑问词+（地方 / 地点 / N / 国家 / 省 / 城市 / 城镇 / 表示地点的名词）	2010 年世界杯在哪里举办 中国的哪个城市最适合人居住
询问实体	通用疑问词+（一般名词）	什么鸟不会飞 奥巴马是哪个党的总统候选人
询问数量	多少、多高、几、多	胡夫金字塔到底有多高 光在真空中的速度是多少
询问方式	怎么 / 怎样 / 如何+（一般动词）、通用疑问词+（方式 / 方法）	地震发生时应该怎么做 什么方法能克服高山反应
询问描述	通用疑问词+（是动词）、怎么样+（一般名词）	什么是温室效应 最近天气怎么样

包含疑问词的问句模板举例：“怎么”这个词，既可能是询问原因的，也可能是询问方法的。“我的包裹怎么没有放行”是询问原因的。“包裹怎么无法退回啊”是询问方法的。

怎么没有 + 动词　　询问原因

怎么无法 + 动词　　询问方法

这里的“怎么没有 + 动词”就是一个模板。根据问句模板对问题分类的代码如下。

```
String question = "我的包裹怎么没有放行";
TypeGrammar grammar = new TypeGrammar();
ArrayList<WordTokenInf> words = grammar.seg(question); //切分问句
```

```
QuestionType questionType = grammar.reduce(words); //匹配问题类型
if(questionType!=null)
    System.out.println(questionType.name); //输出问题类型：询问原因
else{
    System.out.println("null");
}
```

如果答案句中包含时间词，就把问句分类为时间问题。

为了能够找到答案，可以增加额外的代码。为了能够回答“什么动物喵喵地叫”。生成额外的代码如下。

```
public class Cat {
    String say() {
        return "meow";
    }

    boolean isSay(String input){
        return "meow".equals(input);
    }
}
```


8.4 本章小结

知识库构建涉及术语抽取、概念抽取、关系抽取等方法。本章介绍了使用实体对齐技术从顶层创建一个大规模的统一知识库的方法。这样的知识库有 ConceptNet 和 OpenCyc 等。

本章介绍了自动问答系统的实现方法。接下来介绍了其他的自动问答系统。

自动问答系统 Wolfram|Alpha（http://www.wolframalpha.com/）是一个特殊的可计算的知识引擎。它可以根据用户的问句式的输入精确地返回一个答案。如输入：Who is US president? 返回结果：Barack Obama。

和搜索引擎不同，问答系统往往只返回一条整合过的结果，而搜索引擎往往列出多个来源的结果。例如，Wolfram|Alpha 的数据由他们的内部小组策划。有些数据来自官方的公共或私人来源，但大部分是系统的一手信息来源。在返回结果中指出信息源，提供背景来源和参考。

Wolfram|Alpha 后台采用 Wolfram 语言实现。输入 Sunset，返回当天当地的日落时间，输入 Sunrise 和 Today 也有类似的效果。Wolfram 语言调用统一存储在云端的数据。猎兔搜索计划开发支持自然语言输入的编程客户端。输入“东亚国家”，出现包括“中国”这样的国家列表。

TextRunner 搜索（http://www.cs.washington.edu/research/textrunner/）是另外一个问答式的搜索。它使用三元组表示知识。为了改进 TextRunner 中的三元组提取，开发了 ReVerb（https://github.com/knowitall/reverb/）。ReVerb 是一个自动识别和提取英文句子中的二元关系的程序。

ReVerb 将原始文本作为输入，并输出（参数 1，关系短语，参数 2）三元组。例如，给定“Bananas are an excellent source of potassium”这个句子，ReVerb 将提取三元组(bananas, be source of, potassium)。这里的动词作为关系，关联左边的名词短语和右边的名词短语。首先匹配动词短语，然后提取左边和右边的实体。最后验证提取出的三元组。

START（http://start.csail.mit.edu/）也是一个在线问答网站。问林肯是谁，就能回答是美国第几位总统。提交问题：Who is Lincoln。返回结果：Abraham Lincoln, 16th President of the United States。START 是一个支持英语的问答系统。使用自然语言注释技术连接信息查找者和信息来源。这种技术使用自然语言句子和短语注释，用注释来描述内容，注释与信息片断在各种粒度上联系。

当注释匹配一个输入问题时，检索出一个信息片断。给定了知识库中合适的片断后，生成模块就产生了句子。

Elastic 系统监控

可以使用 Elastic stack（Elastic 栈）实现对应用程序日志、事件的传输、处理、管理和搜索。Elastic 栈涉及以下几个组件。

- beats：用于轻量级日志采集，支持文件采集、系统数据采集、特定中间件数据采集等。
- logstash：用于日志结构化、标签化，支持 DSL 方式将数据进行结构化。
- elasticsearch：用于提供日志相关的索引，使得日志能够有效检索。
- kibana：用于提供日志检索，特定 metric 展示的面板，方便使用的 UI。
- x-pack：用于监控与预警相关的组件，可以集成到 Es 中，kibana 有特定的面板用于展示 UI。
- curator：用于管理 Es 集群的索引相关的数据，对索引进行分析。

除了 Elasticsearch，也可以使用 Graylog 管理和搜索日志。本章首先介绍使用 Elastic 栈管理日志，然后介绍 Graylog。

9.1 Logstash

Logstash 是一个收集、处理和转发应用程序事件和日志消息的工具软件。可以用它来统一对应用程序日志进行收集管理，提供 Web 接口用于查询和统计。

收集通过可配置的输入插件完成，包括原始套接字/数据包通信、文件尾部追踪和几个消息总线客户端。一旦输入插件收集到数据后，就可以通过任何数量的修改和注释事件数据的过滤器进行处理。

最后，Logstash 路由事件到输出插件，这些插件可以将事件转发到各种外部程序，包括 Elasticsearch，本地文件和多个消息总线实现。

9.1.1 使用 Logstash

可以用 yum 安装 Logstash。首先在/etc/yum.repos.d/路径写一个 repo 文件指定软件包所在的网址。

```
# vi /etc/yum.repos.d/logstash.repo
```

```
[logstash-5.x]
name=Elastic repository for 5.x packages
baseurl=https://artifacts.elastic.co/packages/5.x/yum
gpgcheck=1
gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch
enabled=1
autorefresh=1
type=rpm-md
```

然后可以用 yum 命令安装：

```
# sudo yum install logstash
```

接下来，通过最简单的方法测试 Logstash 来了解它。

Logstash 管道有两个必需的元素：输入和输出，以及一个可选元素：过滤器。输入插件消耗来自源的数据，过滤器插件会按照指定的方式修改数据，输出插件将数据写入到目的地。

然后到安装目录运行最基本的 Logstash 管道。


```
# cd /usr/share/logstash/bin
# ./logstash -e 'input { stdin { } } output { stdout {} }'
```

在控制台输入 hello world，可以看到类似这样的输出：

```
2017-09-21T08:44:57.280Z iZetooc2z5ucu3Z hello world
```

可以在配置文件中指定输入和输出，一个简单的例子如下。

```
# vi logstash-simple.conf
input { stdin { } }
output {
   stdout { codec => rubydebug }
}
```

启动：

```
# ./logstash -f logstash-simple.conf
```

这里用到了 codec。

根据输入“test”，产生的输出：

```
{
      "@timestamp" => 2017-08-21T10:18:20.749Z,
        "@version" => "1",
             "host" => "iZetooc2z5ucu3Z",
          "message" => "test"
}
```

输出到 Elasticsearch 和 stdout 的配置：

```
input { stdin { } }
```

```
output {
  elasticsearch { hosts => ["localhost:9200"] }
  stdout { codec => rubydebug }
}
```

让 Linux 系统使用 Logstash 最简单的方法是使用传统的日志记录方法：Syslog。

Syslog 是计算机日志记录的原始标准之一。它是由埃里克·阿尔曼（Eric Allman）设计的，是 Sendmail 的一部分，并且已经成长为支持各种日志记录的平台和应用程序。它已成为 Linux 系统上记录日志的默认机制。运行在 Linux 上的应用程序以及打印机和网络设备，如路由器、交换机和防火墙大量使用 Syslog。

Syslog 配置文件位于：

```
# cat /etc/rsyslog.conf
```

默认情况下，RedHat/Fedora 的/etc/rsyslog.conf 文件被配置为将大多数消息放入文件/var/log/messages。Syslog 产生的每个消息的大致结构如下：

```
Aug 28 13:50:50 iZetooc2z5ucu3Z systemd: Starting Session 13738 of user root.
```

消息的组成部分包括：时间戳、生成消息的主机、生成消息的进程和消息的内容。

配置 Logstash 服务器接收 Syslog 消息。Logstash 配置文件位于/etc/logstash/conf.d 目录下。编辑/etc/logstash/conf.d/syslog.conf 文件。

```
input{
    syslog{
    type => "system-syslog"
    port => 514
    }


}

output{
    stdout{
        codec => rubydebug
    }

}
```

启动：

```
# ./logstash -f /etc/logstash/conf.d/syslog.conf
```

9.1.2　插件

通过插件管理器 bin/logstash-plugin 来安装，删除和升级插件。下面列出当前可用的

插件。

```
# ./logstash-plugin list
logstash-codec-cef
logstash-codec-collectd
logstash-codec-dots
logstash-codec-edn
logstash-codec-edn_lines
logstash-codec-es_bulk
logstash-codec-fluent
logstash-codec-graphite
logstash-codec-json
logstash-codec-json_lines
logstash-codec-line
logstash-codec-msgpack
logstash-codec-multiline
logstash-codec-netflow
logstash-codec-plain
logstash-codec-rubydebug
logstash-filter-clone
logstash-filter-csv
logstash-filter-date
logstash-filter-dissect
logstash-filter-dns
logstash-filter-drop
logstash-filter-fingerprint
logstash-filter-geoip
logstash-filter-grok
logstash-filter-json
logstash-filter-kv
logstash-filter-metrics
logstash-filter-mutate
logstash-filter-ruby
logstash-filter-sleep
logstash-filter-split
logstash-filter-syslog_pri
logstash-filter-throttle
logstash-filter-urldecode
logstash-filter-useragent
logstash-filter-uuid
logstash-filter-xml
logstash-input-beats
logstash-input-couchdb_changes
logstash-input-dead_letter_queue
logstash-input-elasticsearch
logstash-input-exec
```

```
logstash-input-file
logstash-input-ganglia
logstash-input-gelf
logstash-input-generator
logstash-input-graphite
logstash-input-heartbeat
logstash-input-http
logstash-input-http_poller
logstash-input-imap
logstash-input-irc
logstash-input-jdbc
logstash-input-kafka
logstash-input-log4j
logstash-input-lumberjack
logstash-input-pipe
logstash-input-rabbitmq
logstash-input-redis
logstash-input-s3
logstash-input-snmptrap
logstash-input-sqs
logstash-input-stdin
logstash-input-syslog
logstash-input-tcp
logstash-input-twitter
logstash-input-udp
logstash-input-unix
logstash-input-xmpp
logstash-output-cloudwatch
logstash-output-csv
logstash-output-elasticsearch
logstash-output-file
logstash-output-graphite
logstash-output-http
logstash-output-irc
logstash-output-kafka
logstash-output-nagios
logstash-output-null
logstash-output-pagerduty
logstash-output-pipe
logstash-output-rabbitmq
logstash-output-redis
logstash-output-s3
logstash-output-sns
logstash-output-sqs
logstash-output-statsd
logstash-output-stdout
```

```
logstash-output-tcp
logstash-output-udp
logstash-output-webhdfs
logstash-output-xmpp
logstash-patterns-core
```

可以使用编解码器插件把事件的数据表示转换成需要的形式。编解码器基本上可作为输入或输出的一部分运行的流过滤器。

安装插件：

```
# ./logstash-plugin install logstash-output-kafka
```

更新插件：

```
# ./logstash-plugin update logstash-output-kafka
```

删除插件：

```
# ./logstash-plugin remove logstash-output-kafka
```


9.1.3 数据库输入插件

MongoDB 输入插件的配置文件。

```
input {
  mongodb {
    uri => 'mongodb://10.0.0.30/my-logs?ssl=true'
    path => '/opt/logstash-mongodb/logstash_sqlite.db'
    collection => 'events_'
    unpack_mongo_id => true
    batch_size => 5000
  }
}
```

这里的数据库文件/opt/logstash-mongodb/logstash_sqlite.db 是自动创建出来的。

应该为/opt/logstash-mongodb 上的 logstash 提供权限，以便它能够创建数据库：

```
sudo chown -R logstash:logstash /opt/logstash-mongodb
```

可以使用 Logstash 同步 mysql 数据到 Elasticsearch。不捕获表上的 DELETE，它们只能捕获 INSERT 和 UPDATE 操作。

从 MySQL 读取数据：

```
input {
  jdbc {
    jdbc_driver_library => "/path/to/mysql-connector-java-5.1.33-bin.jar"
    jdbc_driver_class => "com.mysql.jdbc.Driver"
    jdbc_connection_string => "jdbc:mysql://host:port/database"
```

```
        jdbc_user => "user"
        jdbc_password => "password"
      # or jdbc_password_filepath => "/path/to/my/password_file"
        statement => "SELECT ..."
        jdbc_paging_enabled => "true"
        jdbc_page_size => "50 000"
      }
    }

    filter {
      [some filters here]
    }

    output {
      stdout {
        codec => rubydebug
      }
      elasticsearch_http {
        host => "host"
        index => "myindex"
      }
        }
```


9.2 Filebeat

Filebeat（https://github.com/elastic/beats/tree/master/filebeat）是一个开源文件收集器，可以使用它获取日志文件并将其提供给 Logstash。目前 Filebeat 可以发送数据给 Elasticsearch 或者 Logstash。

在 Linux 下，可以使用 rpm 安装 Filebeat。

```
# curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-5.5.2-x86_64.rpm
# rpm -vi filebeat-5.5.2-x86_64.rpm
```

在配置文件/etc/filebeat/filebeat.yml 中指定了监测的日志路径和输出的目标。如果要输出到 Logstash，可以修改/etc/filebeat/filebeat.yml 相关内容：

```
output:
  logstash:
    hosts: ["localhost:5044"]
```

在 Logstash 的配置文件 logstash.conf 中指定从端口（5044）监听来自 Filebeat 的数据：

```
input {
      beats {
      port => ‘5044’
```

```
        }
    }
```

测试启动：

```
# /usr/share/filebeat/bin/filebeat -e -c /etc/filebeat/filebeat.yml -d "publish"
```

默认的 Elasticsearch 需要的索引模板文件在安装 Filebeat 时已经提供，路径为/etc/filebeat/filebeat.template.json，可以使用如下命令装载该模板：

```
$ curl -XPUT 'http://localhost:9200/_template/filebeat?pretty' -d@/etc/filebeat/filebeat.template.json
```

为了收集 Tomcat 日志，可以修改配置文件/etc/filebeat/filebeat.yml：

```
filebeat.prospectors:

- input_type: log

   paths:
- /install/tomcat/logs/catalina.out
```


设置 filebeat 服务，让它在服务器重新启动时自动启动：

```
# systemctl enable filebeat
# systemctl start filebeat
```

9.3 消息过期

可以使用 Elasticsearch 的 TTL（Time-To-Live）功能定期删除过期的日志信息。

为了让 TTL 工作，必须首先在映射中启用它（默认情况下是禁用的），然后在索引文档时设置 TTL 值。

```
# curl -XPUT 'mybox:9200/blog/user/_mapping?pretty' -d '{
   "user": {
       "_ttl": {"enabled": true}
}'

# curl -XPUT 'mybox:9200/blog/user/dilbert' -d '{ "name" : "Dilbert Brown", "_ttl": "3m"}'

# curl -XGET 'mybox:9200/blog/user/dilbert?pretty'
```

9.4 Kibana

和 Logstash 一样，可以用 yum 安装 Kibana。

```
# yum install kibana
```

启动：

```
# systemctl start kibana
```

可以在本机查看状态：

```
# links http://localhost:5601/status
```

为了能够远程访问 Kibana，修改 Kibana.yml 文件：

```
# vi /etc/kibana/kibana.yml
```

修改 IP 地址：

```
server.host: "0.0.0.0"
```

然后重新启动 Kibana 服务，让配置生效：

```
# systemctl restart kibana
```

这样就可以从远程访问 Kibana 服务了。

增加 Kibana 查询的 Elasticsearch 地址：

```
elasticsearch.url: http://localhost:9200
```


Elasticsearch 将所有 Kibana 元数据信息存储在.kibana 索引下。 Kibana 配置如默认索引和高级设置存储在 index/type/id .kibana/config/4.5.0 中，这里的 4.5.0 指的是 Kibana 的版本。

因此，可以通过以下步骤实现设置或更改默认索引。

（1）将要设置为默认索引的索引添加到 Kibana，可以通过执行以下命令来执行此操作。

```
curl -XPUT http://<es node>:9200/.kibana/index-pattern/your_index_name -d '{"title" : "your_index_name", "timeFieldName": "timestampFieldNameInYourInputData"}'
```

（2）更改 Kibana 配置以将之前添加的索引设置为默认索引：

```
curl -XPUT http://<es node>:9200/.kibana/config/4.5.0 -d '{"defaultIndex" : "your_index_name"}'
```

9.5 Flume

Apache Flume 是一种分布式，可靠和可用的服务，用于高效收集、聚合和移动大量日志数据。它具有基于流数据流的简单灵活的架构。

Flume 事件被定义为具有字节有效载荷和可选的一组字符串属性的数据流的单元。Flume 代理是一个（JVM）进程，它承载事件从外部源传递到下一个目标的组件。可以使用 ElasticSearchSink 将 Flume 采集的数据传输到 Elasticsearch 中。

使用位于 Flume 安装的 bin 目录中的名为 flume-ng 的 shell 脚本启动代理。 需要在命令行中指定代理名称，config 目录和配置文件：

```
$ bin/flume-ng agent -n $agent_name -c conf -f conf/flume-conf.properties.template
```

默认情况下，Flume 将一行视为一个事件。

可以通过 tail 命令监控 Tomcat 日志文件，配置文件内容如下。

```
agent.sources.SrcLog.type = exec
agent.sources.SrcLog.command = tail -F /home/tomcat/webapps/logs/catalina.out
agent.sources.SrcLog.restart = true
agent.sources.SrcLog.restartThrottle = 1000
agent.sources.SrcLog.logStdErr = true
agent.sources.SrcLog.batchSize = 50
```

spooling directory source 监视指定的文件夹下面有没有写入新的文件，有的话，就会把该文件内容传递给 sink，然后将该文件后缀标示为.complete，表示已处理。提供的参数可以将文件名和文件全路径名添加到事件的 header 中去。

ElasticSearchSink 配置如下。

```
agent.sinks.elasticSearchSink.type = org.apache.flume.sink.elasticsearch.ElasticSearchSink
agent.sinks.elasticSearchSink.channel = fileChannel
agent.sinks.elasticSearchSink.hostNames=localhost:9300
agent.sinks.elasticSearchSink.indexName=platform
agent.sinks.elasticSearchSink.indexType=platformtype
agent.sinks.elasticSearchSink.ttl=1m
agent.sinks.elasticSearchSink.batchSize=1000
agent.sinks.elasticSearchSink.serializer=org.apache.flume.sink.elasticsearch.ElasticSearchLogStashEventSerializer
```


9.6 Kafka

Apache Kafka 最初是一个用于日志处理的分布式消息队列，同时支持离线和在线日志处理。Kafka 对消息保存时根据主题进行归类。

要在 Kafka 和另一个系统之间复制数据，用户可以为要从中提取数据或将数据推送到的系统实例化 Kafka 连接器。源连接器将数据从另一个系统（如关系数据库）导入数据。Sink 连接器导出数据（如 Kafka 主题的内容导出到 Elasticsearch 或 HDFS 文件系统）。

Kafka 后来发展成一个分布式流平台。流平台有 3 个关键功能。

（1）允许你发布和订阅记录流。 在这方面，它类似于消息队列或企业消息系统。

（2）允许你以容错方式存储记录流。

（3）可以让你处理记录流。

如果日志写入 Elasticsearch 是系统瓶颈，可以考虑使用 Logstash 从 Kafka 获取数据并将其推送到 ElasticSearch 中。Logstash 配置文件如下。

```
input {
  kafka {
    bootstrap_servers => "localhost:9092"
    topics => ["beats"]
```

```
    }
  }
  output {
    elasticsearch {
      hosts => ["localhost:9200"]
      index => "elasticse"
    }
  }
```

9.7 Graylog

Graylog（https://www.graylog.org/）是一个开源日志管理平台。使用 Elasticsearch 存储日志消息，MongoDB 存储元信息和配置数据。

Graylog Collector Sidecar（https://github.com/Graylog2/collector-sidecar）是第三方日志收集器（如 NXLog）的主管流程。Sidecar 程序能够从 Graylog 服务器获取配置，并将它们转换为各种日志收集器的有效配置文件。可以将它当作对于日志收集器的集中式配置管理系统。

在 Linux 下安装 MongoDB。首先下载：

```
curl -O https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-3.4.4.tgz
```

解压缩：

```
tar -zxvf mongodb-linux-x86_64-3.4.4.tgz
```

在首次启动 MongoDB 之前，需要创建 mongod 进程写入数据的目录。默认情况下，mongod 会将数据写入/data/db 目录。使用以下命令创建该目录：

```
mkdir -p /data/db
```

运行可执行文件 mongod：

```
<path to binary>/mongod
```

创建服务：

```
# mkdir ../log/
# ./mongod --fork --logpath ../log/mongod.log
```

安装 Elasticsearch 版本 1.7。

首先导入用来验证 RPM 包的 RPM GPG 公钥：

```
# rpm --import https://packages.elastic.co/GPG-KEY-elasticsearch
```

添加 Elasticsearch 仓库：

```
# vi /etc/yum.repos.d/elasticsearch.repo
```

```
[elasticsearch-1.7]
name=Elasticsearch repository for 1.7.x packages
baseurl=http://packages.elastic.co/elasticsearch/1.7/centos
gpgcheck=1
gpgkey=http://packages.elastic.co/GPG-KEY-elasticsearch
enabled=1
```

使用 yum 命令安装 1.7 版本：

```
# yum -y install elasticsearch
```

配置 Elasticsearch 在系统启动过程中启动：

```
# systemctl daemon-reload
# systemctl enable elasticsearch.service
```

唯一重要的是将集群名称设置为“graylog2”，这是由 graylog 使用的。现在编辑 Elasticsearch 的配置文件：


```
# vi /etc/elasticsearch/elasticsearch.yml
cluster.name: graylog2
```

禁用动态脚本以避免远程执行，可以通过在上述文件末尾添加以下行来完成。

```
script.disable_dynamic: true
```

一旦完成，就可以重新启动 Elasticsearch 服务以加载修改的配置了。

```
# systemctl restart elasticsearch.service
```

等待至少一分钟让 Elasticsearch 完全重新启动，否则测试将失败。Elastisearch 现在应该监听 9200 端口处理 HTTP 请求，我们可以使用 CURL 来获取响应，确保它返回的集群名称为“graylog2”。

```
# curl -X GET http://localhost:9200

{
  "status" : 200,
  "name" : "Silver Fox",
  "cluster_name" : "graylog2",
  "version" : {
    "number" : "1.7.2",
    "build_hash" : "e43676b1385b8125d647f593f7202acbd816e8ec",
    "build_timestamp" : "2015-09-14T09:49:53Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
}
```

使用以下命令检查 Elasticsearch 集群的运行状况，必须让集群状态是“green”，以使 graylog 正常工作。

```
# curl -XGET 'http://localhost:9200/_cluster/health?pretty=true'

{
  "cluster_name" : "graylog2",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 0,
  "active_shards" : 0,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "delayed_unassigned_shards" : 0,
  "number_of_pending_tasks" : 0,
  "number_of_in_flight_fetch" : 0
}
```


接下来安装 Graylog2。

Graylog-server 接受并处理日志消息，并为来自 graylog-web-interface 的请求产生 REST API。从 graylog.org 下载最新版本的 graylog。

使用如下命令安装 graylog2 存储库。

```
# rpm –Uvh
    https://packages.graylog2.org/repo/packages/graylog-1.2-repository-el7_latest.rpm
```

安装最新的 graylog 服务器。

```
# yum -y install graylog-server
```

编辑 server.conf 文件。

```
# vi /etc/graylog/server/server.conf
```

在上述文件中配置以下变量。

设置一个秘钥来保护用户密码，使用以下命令生成一个秘钥，至少使用 64 个字符。

```
# pwgen -N 1 -s 96
5uxJaeL4vgP9uKQ1VFdbS5hpAXMXLq0KDvRgARmlI7oxKWQbH9tElSSKTzxmj4PUGlHIpOkoMMwjICYZubUGc9we5tY1FjLB
```

如果还没有安装 pwgen，就使用以下命令安装 pwgen。

```
# yum -y install pwgen
```

放置秘钥。

```
password_secret = 5uxJaeL4vgP9uKQ1VFdbS5hpAXMXLq0KDvRgARmlI7oxKWQbH9tElSSKTzxmj4PUGlHIpOkoMMwjICYZubUGc9we5tY1FjLB
```

接下来是为 root 用户设置散列密码（不要与系统用户混淆，graylog 的 root 用户是 admin）。 将使用此密码登录到 Web 界面，管理员的密码不能使用 Web 界面更改，必须编辑此变量才能设置。

用你的选择替换“yourpassword”。

```
# echo -n yourpassword | sha256sum
e3c652f0ba0b4801205814f8b6bc49672c4c74e25b497770bb89b22cdeb4e951
```

放置散列密码。

```
root_password_sha2 = e3c652f0ba0b4801205814f8b6bc49672c4c74e25b497770bb89b22cdeb4e951
```

可以设置电子邮件地址 root（admin）用户。

```
root_email = "itzgeek.web@gmail.com"
```

设置 root（admin）用户的时区。


```
root_timezone = UTC
```

Graylog 将尝试自动查找 Elasticsearch 节点，它使用组播模式。但是，对于较大的网络，建议使用最适合于生产设置的单播模式。因此，将以下两个条目添加到 graylog server.conf 文件中，将 ipaddress 替换为 live hostname 或 ipaddress，可以使用逗号分隔多个主机。

```
elasticsearch_http_enabled = false
elasticsearch_discovery_zen_ping_unicast_hosts = ipaddress:9300
```

通过定义以下变量设置唯一的主节点，默认设置为 true，必须将其设置为 false，以使特定节点作为从属节点。主节点执行一些从属节点不执行的周期性任务。

```
is_master = true
```

以下变量设置每个索引保留的日志消息数，建议使用几个较小的索引，而不是较大的索引：

```
elasticsearch_max_docs_per_index = 20000000
```

以下参数定义索引的总数，如果此数字达到旧索引将被删除。

```
elasticsearch_max_number_of_indices = 20
```

分片设置取决于 Elasticsearch 集群中的节点数，如果只有 1 个节点，将其设置为 1。

```
elasticsearch_shards = 1
```

索引的副本数，如果 Elasticsearch 集群中只有 1 个节点，将其设置为 0。

```
elasticsearch_replicas = 0
```

添加 MongoDB 身份验证信息。

```
mongodb_useauth = false
```

使用以下命令启动 graylog 服务器。

```
# systemctl restart graylog-server
```

可以查看服务器启动日志，如果出现任何问题，这会很有用。

```
# tailf /var/log/graylog-server/server.log
```

在成功启动 graylog-server 后，在日志文件中收到以下消息。

```
2015-09-16T21:26:05.689-04:00 INFO    [ServerBootstrap] Graylog server up and running.
```

安装 Graylog 网页界面：

要配置 graylog-web 界面，必须至少有一个 graylog-server 节点。使用以下命令安装 Web 界面。

```
# yum -y install graylog-web
```


编辑配置文件并设置以下参数。

```
# vi /etc/graylog/web/web.conf
```

这是 graylog-server 节点的列表，可以添加多个节点，用逗号分隔。

```
graylog2-server.uris="http://127.0.0.1:12900/"
```

设置应用程序 scret，可以使用命令 pwgen -N 1 -s 96 生成它。

```
application.secret="sNXyFf6B4Au3GqSlZwq7En86xp10JimdxxYiLtpptOejX6tIUpUE4DGRJOrcMj07wcK0wugPaapvzEzCYinEWj7BOtHXVl5Z"
```

使用以下命令重新启动 gralog-web 界面。

```
# systemctl restart graylog-web
```

访问 Graylog Web 界面。

Web 界面将侦听端口 9000，配置防火墙以允许端口 9000 上的流量。

```
# firewall-cmd --permanent --zone=public --add-port=9000/tcp
# firewall-cmd –reload
```

将浏览器指向 http://ip-add-ress:9000。使用用户名“admin”和在 server.conf 文件中 root_password_sha2 选项指定的密码登录。

9.8 物联网数据

可以将传感器设备嵌入到现有的消息代理、日志数据处理和可视化基础架构中。消息代理处理的传输数据不仅可以由企业系统生成，而且还可以由物联网（IoT）设备和应用的

计算单元生成。

单板计算机树莓派（Raspberry Pi）——一个网关，负责初始处理从传感器通过 GPIO 提交的数据，并通过 MQTT 向消息代理发送数据。树莓派具有通用输入/输出（GPIO）和 USB 接口，可以通过连接的设备灵活扩展，包括能够收集各种计量和遥测信息的设备。安装和运行采用的完整 Linux 系统的可能性为执行已发布和定制开发的程序（以 Python、Java、Ruby、Perl、C/C ++和其他一些语言编写）提供了必需的运行时和环境。这里使用树莓派设备上的 Raspbian OS（基于 Linux Debian 的 Raspberry Pi 版本）的正式发行版本和开发脚本的 Python。

Raspberry Pi Sense HAT（Hardware Attached on Top）——传感器块是安装在树莓派主板上的扩展板。

RabbitMQ——消息代理。 RabbiMQ 最初是一个在异构 IT 环境中流行的 AMQP 消息代理，并且有效地用于企业应用集成领域以及 IT 基础设施组件之间的消息排队和交换，其中关键方面是高性能和吞吐量，可扩展性和高可用性以及综合消息生产者和消费者的技术多样性。

这里使用消息队列遥测传输（MQTT）协议，这是一种机器对机器（M2M）连接中的事实上的协议之一，其随着物联网概念的演进和普及而普及。MQTT 协议可以被认为是高级消息队列协议（AMQP）的轻量级和简化模拟，也是基于发布/订阅模式，但提供了更小的网络足迹，并且可以被小型设备利用。

树莓派连接到 Wi-Fi 网络以简化对它的访问，RabbitMQ 和 ELK 工具安装在另一台机器上的虚拟化环境中。虽然技术上可以安装在树莓派本身上，但这里的目的是在逻辑上将“IoT”部分从消息代理和数据收集、处理及可视化工具等中心组件分离。

可以开发一个 Python 脚本，以便从传感器块收集遥测信息（在这里，仅收集温度、湿度和压力的指标），并将其发布到通过 MQTT 协议在消息代理处注册的主题中。

可以在树莓派引进温度、湿度以及烟雾的传感器，监测医院自助机，以防止因为温度过高而导致蓝屏。

9.9 本章小结

Elasticsearch 与数据收集和日志解析引擎 Logstash 以及分析和可视化平台 Kibana 一起开发。 这三款产品被设计成集成的解决方案，称为“Elastic 栈”（以前称为“ELK 栈”）。

Splunk 为来自任何应用、服务器或网络设备（包括日志、配置文件、消息、警报、脚本和指标）的数据实时建立索引并使其变得可搜索。ELK 提供了日志分析的开源实现，比 Splunk 更节约成本。

Logstash 项目诞生于 2009 年 8 月 2 日。其作者是世界著名的运维工程师乔丹西塞（JordanSissel），乔丹西塞当时是著名虚拟主机托管商 DreamHost 的员工，还发布过非常好的软件打包工具 fpm，并主办着一年一度的 Sysadmin Advent Calendar。

乔丹西塞曾经在博客上承认设计想法来自 AWS 平台上最大的第三方日志服务商 Loggy，而 Loggy 两位创始人都曾是 Splunk 员工。

2013 年，Logstash 被 Elasticsearch 公司收购，ELK stack 正式成为官方用语（后来正

式命名为 Elastic 栈）。Logstash 水平可伸缩的数据处理管道与强大的 Elasticsearch 和 Kibana 协同作用。

Elasticsearch Curator 可以帮助你策划或管理你的索引。

Flume 是分布式的日志收集系统。

本章介绍了 ELK 日志分析系统的集群搭建。为了实现数据备份，可以搭建 Hadoop 平台，并存储 ELK 的历史数据。

Graylog 是一个开源日志管理平台，也使用了 Elasticsearch 存储和搜索日志。可以使用 Elasticsearch、Graylog 和 MongoDB 构建分布式日志服务器。

企业的高管们不想再看到另一个仪表板，他们想读到一个可操作的文字段落。在将来，自然语言生成技术会更有效地利用日志信息。

反侵权盗版声明